ELECTRONIC ENGINEER

C/C++ Language Programming Methods for TMS320C5000 DSP

C/C++语言硬件程序设计

——基于TMS320C5000系列DSP

[redacted]

C/C++语言硬件程序设计

——基于 TMS320C5000 系列 DSP

张　勇　编著

陈天麒　主审

西安电子科技大学出版社

2003

内 容 简 介

本书全面介绍了使用 C/C++高级语言开发 TMS320C5000 系列 DSP 应用系统的方法。全书共分为七章，内容包括 TMS320C5000 系列开发平台、TMS320C5000 系列硬件基础、C/C++程序设计、DSP/BIOS 程序设计、RTDX 程序设计、Boot Loader 程序设计和应用实例。本书的特色在于通过完整的程序实例介绍相关的内容。

本书是作者近几年来不间断地从事 TMS320 系列 DSP 系统设计和开发的技术结晶，书中偏重于讲解用 C/C++语言对 DSP 的硬件资源访问。本书内容除具有 TMS320C5000 系列 DSP 程序设计的通用性外，还对 TMS320C6000 系列 DSP 程序设计具有直接的指导意义。

本书可以作为从事 DSP 开发的电子工程技术人员以及 DSP 爱好者的参考书，也可以作为高校电子类相关专业的学生学习 DSP 及其程序设计的参考书或教科书。

C/C++ 语言硬件程序设计
——基于 TMS320C5000 系列 DSP

张 勇 编著
陈天麒 主审

责任编辑 李惠萍
出版发行 西安电子科技大学出版社（西安市太白南路 2 号）
电 话 (029)8242885 8201467 邮 编 710071
http://www.xduph.com E-mail: xdupfxb@pub.xaonline.com
经 销 新华书店
印 刷 西安兰翔印刷厂
版 次 2003 年 5 月第 1 版 2003 年 5 月第 1 次印刷
开 本 787 毫米×1092 毫米 1/16 印张 16.25
字 数 383 千字
印 数 1～4 000 册
定 价 22.00 元
ISBN 7 – 5606 – 1190 – 7 / TP・0619
XDUP 1461A01-1

前言

随着微电子技术、计算机技术和通信技术的迅猛发展，数字化技术已广泛深入地应用于现代国防、现代科技和国民经济的各个领域，在社会活动和个人生活中都随处可见其形影。数字化技术的基础和核心是通用数字信号处理器（DSP）及其相应的程序软件。这就使得DSP及其应用程序在科技、国防、国民经济、社会和个人生活中占有特殊地位。因此，开发应用DSP及其相应软件是当今科学和社会发展所需。

DSP芯片的最大优越性在于其可重复编程的能力。将各种不同应用的数字信号处理技术以软件的形式下载到DSP芯片中，可以实现相应的通信和控制功能。DSP的发展和应用使得软件开发和硬件设计变得相对独立，DSP是数字信号处理硬件系统的核心，DSP硬件系统具有灵活的可编程性。C/C++语言因其具有灵活、易学和可访问硬件资源等特性，已成为DSP软件设计的最佳高级语言。所以，DSP硬件知识（包括CPU、存储器、片上外设等）与C/C++程序设计语言是进行DSP系统设计的最基础的知识。

DSP的开发分为两部分：一部分是硬件平台的设计，另一部分是软件平台的设计。前一部分设计追求的是一定范围内的广泛通用性，后一部分设计追求的是软件设计的模块性。硬件平台的设计一方面要求使用通用数字信号处理器或它们的并行处理器作为整个硬件平台的核心，这是一条最本质的原则；另一方面，通过借用一个或几个现场可编程逻辑器件如CPLD、FPGA等完成一些控制或缓冲，平台上具有高速的可编程数字编码器件（如高速A/D，D/A等），具有一些可以长期存放数据的存储器，还应提供灵活的接口和美观的人机界面等。软件的设计，不是通常意义上的计算机软件设计，它更注重高效、实时等特性。

本书主要介绍使用C/C++语言对DSP硬件进行编程的方法，主要内容分为7部分，依次介绍了开发平台建设、硬件基础、C/C++语言程序设计、DSP/BIOS、RTDX和在线Boot Loader，以及一个完整实例。本书内容是作者进行DSP系统开发和研究成果的结晶。本书选取了TI公司的高速低功耗的TMS320C5000系列DSP为介绍对象，以TMS320VC5402为例，系统地介绍了DSP的硬件资源和使用C/C++语言完成DSP系统的程序设计的具体过程。其中，重点突出了C/C++语言与DSP硬件相结合的编程方法与技巧。通用数字信号处理算法的C/C++语言程序设计，与在计算机上开发C程序类似，本书仅以一个语音自动增益控制为例作了方法上的介绍。

本书的最大特点在于作者给出了进行DSP程序设计的完整实例和程序代码。这些实例是以基于SY-5402EVM板上的语音信号处理模块的形式

出现的，具有代表性和参考价值。C/C++语言作为高级语言在各种 DSP 平台上具有很好的可移植性，基于 C/C++语言的 CCStudio 编译支持技术不断提高，进行模块化的 C/C++语言程序设计将有望取代汇编语言进行 DSP 程序设计。

本书在硬件上为读者展示了 TMS320VC5402 等芯片，在软件设计上为读者展示了用 C/C++语言进行通用 DSP 硬件开发的过程及应用实例。虽然，无论是硬件和软件设计，本书的重点都放在 TMS320VC5402 的介绍上，但是它们对 C5000 系列甚至 C6000 系列都有直接的通用性。一方面，C5000 系列具有相同的 CPU 核（严格说来，C5400 和 C5500 的 CPU 核在总线个数和运算速度上有所区别），具有大体相同的外设接口和大小不同的存储空间。因此，对它们的 C/C++语言程序设计是完全通用的；对它们外设的访问方法也是相同的，只是在硬件设计上要根据不同的要求选用具有不同外设和不同存储空间的 DSP。也就是说，本书是面向 C5000 系列的一本程序设计书，本书的所有程序几乎不需要修改就可以直接移植到其它 C5000 系列 DSP 上，读者可以举一反三。另一方面，C6000 系列是 TI 公司推出的目前运算速度最高的 DSP 系列，C6000 包括 C62xx、C64xx 和 C67xx；C6000 和 C5000 只有一个本质的区别，这也是 C6000 系列的本质所在，那就是 C6000 的“并行”思想。对于一个掌握了 C5000 的读者来说，严格意义上讲，只要具备了“并行”思想，基本上已经初步理解 C6000 了。这个“并行”思想，体现在两个方面：一方面，体现在硬件结构上，并集中体现在 CPU 内部；另一方面，体现在软件设计上，且集中体现在对 C/C++语言或是汇编语言的编译、执行过程中。一个 CPU 时钟为 200 MHz 的 C6000，当每个时钟周期并行 8 条指令时，即一个 C6000 相当于 8 片 C5000 并行使用，运算速率为 1600 MIPS（百万条指令每秒）。另外，C6000 也支持 ANSI C 和 C++语言，本书的 C/C++语言介绍直接适用于 C6000；C6000 的 DSP/BIOS 和 RTDX 与 C5000 的相比是大同小异的；C6000 的 Boot Loader 比 C5000 的 Boot Loader 方便些。

本书可以作为从事 DSP 开发的电子工程技术人员的参考书，也可以作为电子类相关专业的本科及硕士生学习 DSP 开发应用或进行毕业设计的参考书。为了对每一章的学习作一个自我测试，每章后面都附有习题，这些习题是对本章内容的一个总结，也是为了强调本章需要熟练掌握的知识。如果本书有幸成为高校相关专业课的教科书，建议对本书的程序也进行一定的讲解，并需要结合硬件平台给学生作详细的分析。如有必要，您可以通过 EMAIL（zhnyong1975@163.com）或 OICQ（105351369）与作者交流。

学习本书可以按章节顺序进行，也可以先学第二章的部分内容。此外，建议读者登录到 www.ti.com 网站，了解一下 C5000 最新的动态及应用范围。国内上海三意电子公司的官方网站 www.dspdsp.com 上，也有很多 C5000 的技术文档可供下载。

编著者

2003 年 3 月于电子科大

目 录

第一章

TMS320C5000系统开发平台

1.1 本章内容简介

本章介绍使用C/C++语言开发TMS320C5000（下文简称C5000）系列DSP应用系统软件所必须配备的硬件设备和软件平台。一般来说，开发C5000系列的常用硬件设备是计算机、DSP仿真器、DSP功能板以及相应的各种信号源、能量源和测试设备。开发C5000系列的软件平台是TI公司推出的Code Composer Studio（下文简称CCStudio），它可以运行于各种Windows操作系统上。

本章分为三部分，首先介绍开发DSP应用系统的过程，然后，在“开发平台建设”中详细地阐述了C5000系列的硬件和软件开发环境，在“CCStudio初步探索”中对CCStudio的集成开发环境（IDE）作了详细介绍。通过这部分的学习可以使读者熟悉整个开发环境，为后面学习应用C5000系列芯片进行C/C++语言程序设计奠定基础。

思考题

(1) 如何使用CCStudio的Visual Linker?
(2) 如何使用CCStudio的Setup?
(3) 如何测试C/C++程序的执行时间?

1.2 开发 DSP 应用系统的过程

在 DSP 应用系统设计上，一个优秀的开发过程是符合“硬件设计的软件化，软件设计的硬件实现”这一原则的，特别是大型项目的开发，对于算法的评估不应仅限于计算机的仿真实现上，这样离 DSP 应用系统开发过程太远。并非基于通用计算机软件的仿真对开发 DSP 系统没有任何指导作用，但却不符合开发 DSP 应用系统的原则，而且也不是不可被替代的分析过程。

开发 DSP 应用系统的总体过程如下：

? 第一步，系统工程师要对开发的项目作总体分析，对项目所实现的功能和所有实现这些功能的技术作详细的描述，对项目未来的功能扩展、发展趋势和应用前景作总结报告，对于使用 DSP 系统实现的可行性作初步的预测，并给出项目开展的大体方案。

? 第二步，软件工程师根据项目的需求选用合适的 DSP 系统 EVM 板。DSP 系统 EVM 板是 DSP 生产商或代理商开发的用于评估数字信号处理算法的硬件电路板卡。此外，EVM 板还可以进行功能扩展，并直接应用到具体项目中。也就是说，EVM 板将会节约项目开发的时间并减少了资源浪费，因此，在实际开发 DSP 应用系统时，EVM 板是必需的。软件工程师可以借助 DSP 系统 EVM 板进行算法设计、调试、评估和可行性分析。在此基础上，软件工程师就项目开发所用的数字化技术作设计和实现上的技术可行性报告。需要说明的是，EVM 板也是学生学习和掌握 DSP 应用系统及其编程方法的有力工具。

? 第三步，硬件工程师根据项目对硬件的具体要求，在 DSP 系统 EVM 板的指导下，设计出本项目所专用的通用硬件 DSP 系统平台，并作设计方案及设计成品的总结报告。

? 第四步，参与项目开发的所有工程师，将程序移植到项目的 DSP 系统平台上，进行样机各项指标的性能测试，并做出项目的详细技术文档备案，作项目的评审、总结和投产报告。

本书重点讲述了上面介绍的第二步。对于软件工程师和高校学生来说，设计面向 DSP 应用系统的软件，必须对 DSP 芯片的内部资源有详细的了解，特别是存储器配置、片上外设访问和中断向量表的构造；应掌握面向 DSP 开发的 C/C++语言程序设计。关于基于 DSP 系统 EVM 板的整个软件开发平台的学习可以参考本书第 1.3 节“开发平台建设”和第 2.4 节“SY-5402EVM 板”；关于 DSP 芯片的内部资源介绍可以参考第 2.2 节“TMS320VC5402 简介”和第 2.3 节“TMS320VC5510 简介”；关于开发 DSP 系统的专用软件 CCStudio 的学习可以参考第 1.4 节“CCStudio 初步探索”；关于编写存储器配置文件和中断向量表的方法可以参考第 2.4 节“SY-5402EVM 板”;关于 C/C++语言程序设计的学习请参考第三章“C/C++程序设计”。

学习 DSP 编程大体上可以分为以下几步来循序渐近地完成：

? 第一步，读者应该首先登陆到 http://www.ti.com 和 http://www.dspdsp.com 等介绍 DSP 的专业网站去了解 DSP 的总体情况，了解 DSP 芯片已经使用的领域和未来发展的领域。通过这些认识，可以对自己是否需要学习 DSP 编程和开发起到指导和导航作用。在这些网站上也有大量 DSP 开发的详细技术资料和一些免费的软件库可以下载，同时还公开了一些新的开发标准。

？第二步，充分地了解 DSP 的内部资源是熟练学习 DSP 编程的基础。本书讲述使用 C/C++语言的编程方法，该方法易学难精，因此必须全面把握 DSP 的内部资源情况才行。对于汇编语言，读者没有必要去学习其编程方法，但应该搞懂汇编语言的格式和基本的寻址方法，能借助“帮助（Help）”信息读懂汇编语言。这些很容易做到，因为汇编语言基本上是见名知义的算符或助记符。在 CCStudio 环境下，选中或把光标移动到汇编语言指令上，按下 F1 键就会弹出该命令的解释。作者强调对汇编语言的了解是从并行处理分析的角度出发的，可以为调试 C/C++语言程序提供方便。

？第三步，全面学习并掌握面向 DSP 的 C/C++语言的数据结构、语法、库函数、编程方法等等，培养一种良好的编程思想和编程习惯。面向 DSP 的 C/C++语言程序设计有它自身的特点，不在注重程序界面的高贵典雅，它主要是面向实时处理的程序设计，只重视高效性、实用性、简洁性，有时还要求并行性等，重视转化为汇编语言后的最优化性等等。因此学习面向 DSP 的 C/C++语言程序设计时，要有一种全新的编程思想。首先，程序是处于“死循环”状态的，普通的算法程序是运行完毕后就会退出，由 Windows 等操作系统管理资源并处于“死循环”状态。DSP 程序的“死循环”并不是一种死机的状态，它是一种空闲（idle）状态，是一种等待新的指令的状态。打破这种 idle 状态的方法有很多种，常用的是中断处理和循环访问的方法。中断是 DSP 正常工作的灵魂。其次，对于程序的输入输出也要有一个新的认识。普通计算机 C/C++语言编程的输入是来自于键盘、鼠标或是文件的输入，输出是到显示器或打印机等一些人们习以为常的设备上；而 DSP 的 C/C++语言程序的输入和输出都是针对它的存储空间来说的，即输入来自存储器的某个或某些地址，输出结果送到存储器的某个或某些地址。在调试程序方面，除了通用的 C/C++语言程序调试方法外，还有专用于 DSP 的调试方法，如观测 CPU 寄存器等。学习 DSP 的 C/C++语言编程方法，单靠一台计算机是不够的，最好是能够借助 DSP 的各种 EVM 功能板和仿真器等进行编程。

？第四步，学习一种硬件全部抽象的 C/C++语言编程方法，即 DSP/BIOS 编程方法。这种编程方法依然需要对硬件资源的充分了解，但因其把 DSP 的硬件资源全部抽象为句柄，所以该方法是一种在 C5000 系列平台上具有强可移植性的编程方法。这种编程方法也提供了一种实时调试 DSP 程序的方法，是分析和测试 DSP 算法的强有力的工具。

？第五步，学习用于计算机和 DSP 进行实时通信的 RTDX 编程方法，也可以称为 RTDX 技术。在 RTDX 技术支持下，可以通过 JTAG 接口在计算机和 DSP 之间实现数据的实时通信功能。DSP 应用系统设计的实质在于实现主机应用程序和目标机应用程序之间数据的通信。例如，使用 RTDX 技术可以在不干预 DSP 正常工作的情况下实时捕获 DSP 中的数据和事件数据，了解 DSP 的运行情况，也可以由 DSP 的应用程序通过 RTDX 与计算机应用程序交换测试数据。RTDX 的重要性在于测试的灵活性和实时性。

？第六步，学习 DSP 的算法标准，编写出高效的代码库，达到精通 DSP 编程的目的。

？第七步，学习 DSP 的 Boot Loader 编程方法，学会生成 Boot 程序以及利用 CCStudio 在线写 FLASH 存储器等的方法。

本书也是以上面的几个步骤来编排的（第六步省略掉了）。通过本书的学习将使读者具有一定的基本功。这只是一个先决条件，只有不断地从事 DSP 的算法研究和编程开发才能最后圆读者一个精通 DSP 的 C/C++编程的梦。

1.3　开发平台建设

1.3.1　硬件设备

1) 准备硬件环境

在编写第一个面向 DSP 的 C/C++语言程序之前，搭建编程的硬件环境是必须先行的。这个硬件环境由计算机、DSP 仿真器和 DSP 功能板组成，条件好的还可以包括一些测量仪器。就当前的水平来说，采用低价的具有 1 GHz 的 CPU、256 MB 的内存、20 GB 的硬盘和 1024×768 分辨率显示器的计算机已经是足够用的了，建议安装 Windows 2000+SP3 操作系统。DSP 仿真器分为 XDS510 和 XDS560 两种型号，又分别支持并口、ISA、PCI、LAN 或 USB 等计算机接口，可根据自己项目的需要和资金情况选用合适的设备。作者是选用了闻亭公司的 EPP_XDS510 仿真器（来自三意电子）。DSP 功能板一般是选购 TI 或其代理商生产的 EVM 板。对于开发 C5000 系列的产品来说，因为 C5400 系列具有相同的 CPU 结构，C5500 系列也具有相同的 CPU 结构，所以可以根据项目开发所需要的片上外设和 CPU 的速度选用相应的 EVM 板卡。作者使用了上海三意电子公司赞助的 SY-5402EVM 板，如图 1-1 所示。关于 SY-5402EVM 板的详细说明请参考第 2.4 节“SY-5402EVM 板”。

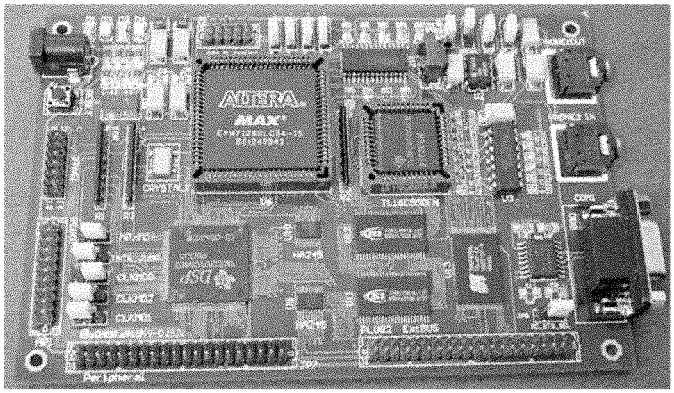
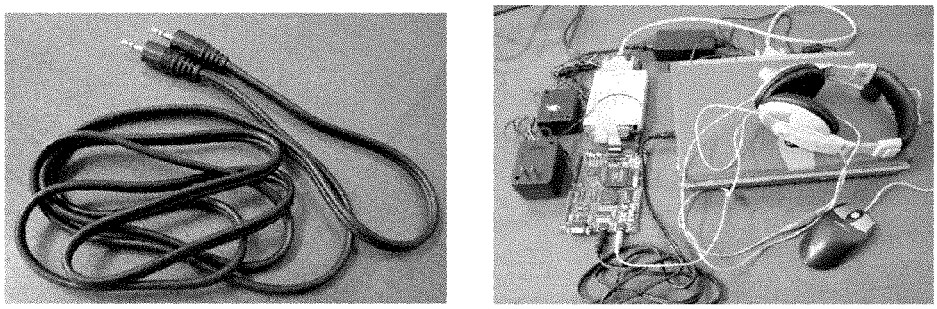


图 1-1　SY-5402EVM 板

2) 搭建硬件平台

搭建硬件平台的具体方法是：关闭计算机，将 EPP_XDS510 仿真器通过并口电缆连接到计算机的并口上，拧紧螺栓，注意不要带电插拔（作者试过仿真器不上电时的开机带电插拔并没有严重影响，但不建议这样做）；将+5V 电源变压器的输出头连接到仿真器上，仿真头连接到 SY-5402EVM 板上，SY-5402EVM 板接头处连接上+5V 电源的接头（SY-5402EVM 板的+5V 电源越稳定越好）；打开计算机电源，将 SY-5402EVM 板和仿真器的电源插头接好，这三者的顺序无关紧要。一切工作就绪后，可以看到在 EPP-XDS510 仿真器上的两个指示灯亮起，一个是红色灯（POWER），指示仿真器本身电源供电情况，另

一个是绿色灯（TARGET），指示目标板的连接情况。作者为了测试程序的运行情况还准备了一条能与计算机音频信号输出端相连接的音频线和一个耳机。全部硬件设备如图 1-2 所示，总的投资在万元以内，适合于学生学习和电子软件工程师开发应用。



(a)　　(b)

图 1-2　实验用硬件平台

(a) 音频线；(b) 总的实验硬件平台

补充一点，上面借助于仿真器和 EVM 板即借助于硬件 DSP 芯片的开发环境称为仿真环境（Emulator），CCStudio 也提供了设计 DSP 程序的计算机模拟环境（Simulator）。在模拟环境下，只需要一台计算机即可，但收效较慢，意义不大，而且程序只是模拟运行。可以通过调试器观察程序运行所用的指令周期来评估算法。真正下载到实际 DSP 功能板上时需要作一些调整。建议初学者可以用这种方法了解一下 CCStudio 环境以及进行 DSP 算法编程。

1.3.2　软件平台

学习和开发 DSP 应用系统的 C/C++语言程序设计必备的软件是 TI 公司的 Code Composer Studio（简称 CCStudio），以及支撑它的计算机操作系统，例如 Windows 2000+SP3 等。

CCStudio 是 TI 公司开发的专用于进行 TMS320 系列 DSP 软件设计的集成软件开发环境。CCStudio 提供了丰富的在线“帮助”文件和强大的在线“帮助”功能，并提供大量的 PDF 格式后台“帮助”文件，通过这些“帮助”文件用户可以深入地了解 DSP 软件设计的方法和过程。CCStudio 提供了图形化的编辑、编译、汇编、连接和调试环境以及友好熟悉的操作界面。CCStudio 通过安装不同的编译库可以用于支持不同系列的 DSP，例如，安装了 TMS320C55x 编译连接库，CCStudio 平台就可以支持 TMS320C55x 系列 DSP 的编程开发。值得一提的是，CCStudio 中集成了 ANSI C 的全部编译支持和大部分 C++的编译支持。面向 DSP 的 C/C++程序设计更注重效率，所以不一定非要使用 C++来展示一下编程的能力。

设计者的 C/C++语言源程序在 CCStudio 集成环境下生成一个可执行的目标文件或可以下载的目标文件，最终可将这个目标文件下载到 DSP 功能板上的 FLASH 中，使得 DSP 功能板可以脱离仿真环境独立工作。在仿真环境下的工作原理是这样的：首先，通过 CCStudio 将 C/C++程序编译连接成一个目标文件；其次，在 CCStudio 环境下使用装入命令通过仿真器和 JTAG 接口把目标代码写入到 DSP 芯片的映射存储器中；再次，使用 CCStudio 运行菜

单向 DSP 芯片发送运行指示，这时 DSP 芯片会执行映射存储器内的程序；最后，可以通过 CCStudio 暂停或停止 DSP 内正在运行的程序来观测 DSP 芯片内部的资源情况和运行的中间结果。因此，仿真过程中程序是在 DSP 芯片上运行的，CCStudio 起到一个对运行过程遥控管理的作用；而模拟过程中，程序是在计算机上运行的。细心的读者会注意到同一个程序在仿真环境下运行的速度远远高于在模拟环境下的运行速度（这是作者使用了软件无线电中的正交解调算法进行测试的结果），这就是为什么实际开发实时 DSP 算法时不能借助于模拟器（Simulator）的主要原因，特别是当需要对 A/D/A 送来的数据进行实时处理时更应如此。

现在假设读者具备了硬件和软件平台，且硬件已经上电复位，CCStudio 软件已经安装成功（作者使用了 CCStudio2，安装方法按提示进行，建议全部安装，本书中不再提及），下面的工作就是如何将软件和硬件平台结合起来，构成一个合法的仿真平台，以及设计硬件和软件的初步使用方法，请继续阅读下一节的内容。

1.4　CCStudio 初步探索

1.4.1　仿真器的驱动程序的安装

这里仅就 XDS510PP 系列仿真器的安装作一个简单的说明。

可以先安装 CCStudio，再安装 XDS510PP 的驱动程序，也可以先安装 XDS510PP 的驱动程序，再安装 CCStudio，这个顺序并不重要。读者甚至可以在没有连接 XDS510PP 仿真器的情况下，先将 CCStudio 和驱动程序安装好。针对 C5000 和 C6000 系列有两种不同的驱动库，读者可以都安装到计算机上，但是，使用 SETUP 配置时，要针对具体的 DSP 功能板上的芯片设置成相应的驱动库。

安装完 CCStudio 和仿真器驱动程序后，读者应重新启动计算机进入 BIOS，设置并口模式为 EPP，端口地址设为 0x378。进入 Windows 环境下后，可以进入设备管理器查看一下并口的设置，如图 1-3 所示。

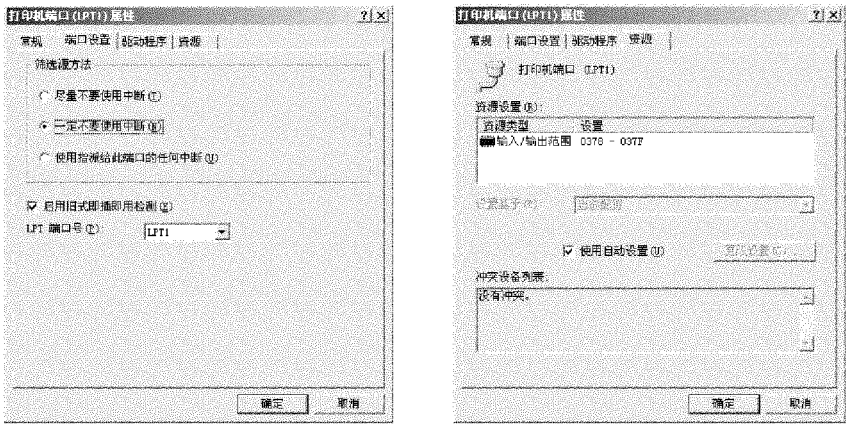


图 1-3　打印机端口的设置（Win2000+SP3 下）

如果使用 Win2000，用户必须以管理员身份登录。假设用户将仿真器的驱动程序安装在 C:\ti 下面，则进入 DOS 环境的 C:\ti\specdig\bin 目录，执行 portio –e 进行并口测试，成功后，运行 portio –s 会在当前目录下自动生成一个文件 xds510pp.sav。对于 Win98、Win95 用户，应将这个文件改名为 xds510pp.ini 存入 C:\windows\system 目录中；对于 Win2000 用户，应将这个文件改名为 xds510pp.ini 存入 C:\winnt\system32 目录下。这个文件的内容为：

```
//This file is automaticly
//generated by portio.exe -s
port    = 378
mode    = EPP
speed = 0
```

点击桌面上的 SDConfig 图标，确定如图 1-4 所示的配置。一般情况下，默认的设置不需要修改。

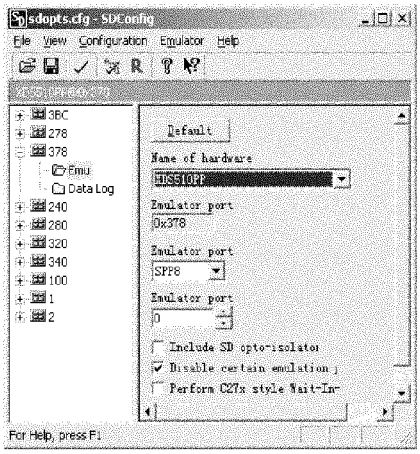


图 1-4　SDConfig 配置情况

上面这些测试应在计算机并口、仿真器和 DSP 功能板均连接正确且上电后操作，适当的时候还要重新启动计算机。

1.4.2　Setup 使用简介

上一节中我们仅是将计算机并口、仿真器和 DSP 功能板之间的电气连接建立起来了，但是还不能进行通信仿真。CCStudio 可以对任一 DSP 芯片或其组合形式进行仿真，具体要仿真的 DSP 芯片类型由实际 DSP 功能板上的 DSP 芯片决定。通过 CCStudio 的 Setup 程序来设置仿真 DSP 芯片的驱动程序库，从而可以对相应的 DSP 芯片进行仿真分析。也就是说，Setup 程序只有一个作用，即用于设置 CCStudio 所需要仿真的 DSP 芯片的驱动程序，使得 CCStudio 可以对该 DSP 芯片进行仿真。

CCStudio 安装成功，后会在桌面上呈现如图 1-5 所示的图标。Setup CCS2('C5000)即为 Setup 程序，另一个图标 CCS2（'C5000）为 CCStudio 的集成开发环境（IDE）。点击 Setup CCS2 图标进入如图 1-6 所示的界面。

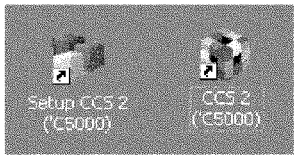


图 1-5　CCStudio 的桌面图标

在图 1-6 的小对话框中，向用户展示了 TI 公司提供的可利用配置，包括 Emulator 和 Simulator 两种；下面的过滤器与上面的可选配置列表框是“粘连”的(“相关”的)，可以通过过滤器将目标配置适当减少以便于用户选择；最下面的复选框选中后，每次打开 Setup 都会弹出该对话框。点击 Clear 按钮清除“System Configuration”栏中的所有项，关闭这个小对话框。点击“Install a Device Driver”选取“sdgo5xx.dvr”，“sdgo5xx”会出现在“Available Board/Simulator Types”中，将“sdgo5xx”拖到“System Configuration”栏中，出现图 1-7 所示界面。

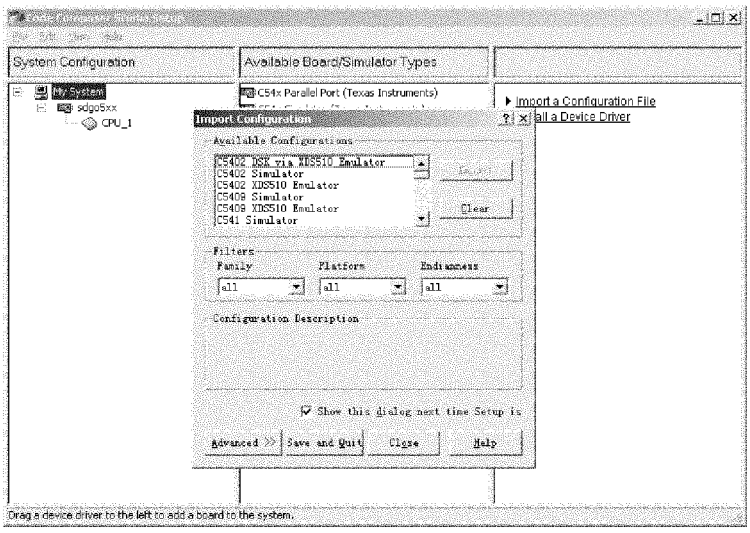


图 1-6　CCStudio 的 Setup 程序界面

图 1-7 中有 4 个选项卡，在仿真环境下不用设置“Startup GEL file(s)”选项卡，因为即使设置了也没有太大的意义。虽然 GEL 文件也可以对借助仿真器和 JTAG 接口对 DSP 的资源进行一些配置，但是这对于目标程序的运行是毫无意义的；GEL 的重要作用在于计算机模拟环境下，对计算机模拟 DSP 芯片的模拟环境进行一次初始化操作，满足用户在模拟环境下的各种硬件环境的设定。因此，作者建议在仿真环境下不要加入 GEL 文件，有了这个 GEL 可能会弄巧成拙的。为了适应一些模拟用户，我们在第 1.4.4 节介绍了 GEL 语言。

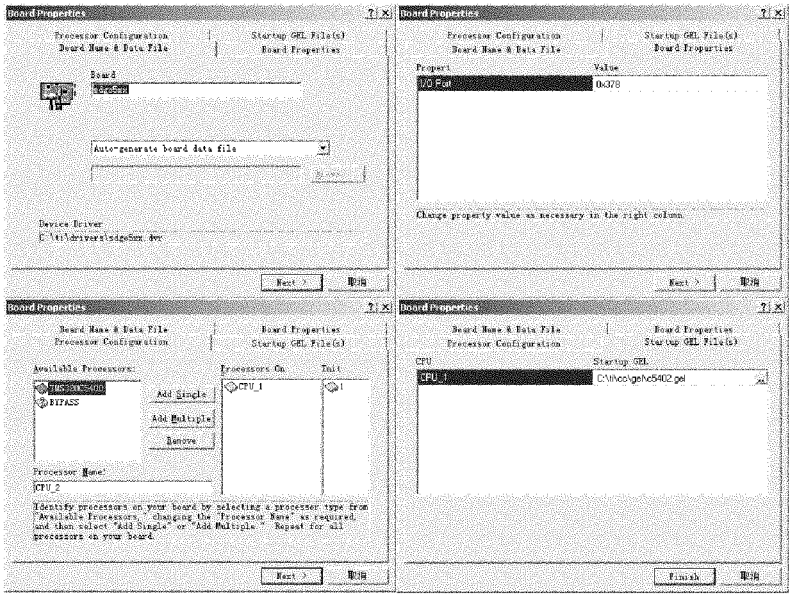


图 1-7　DSP 功能板的设置

单击 Finish 按钮进入到图 1-8 所示的窗口，使用“File”菜单下的“Save”命令存储配置，然后退出 Setup 环境。退出 Setup 时会询问用户是否启动 CCStudio，点击“是”，立即进入 CCStudio 环境；或单击“否”，退到 Windows 桌面上，双击“CCS2”图标，也可进入 CCStudio 环境。

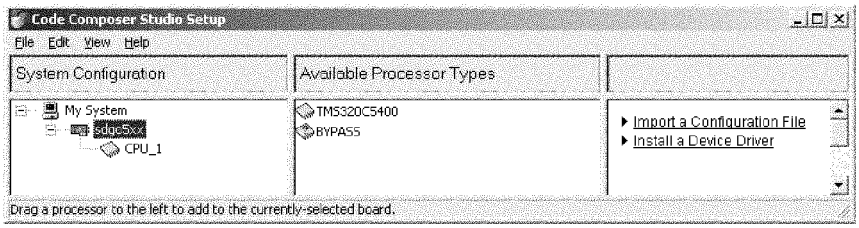


图 1-8　Setup 配置的最终结果（针对 XDS510PP）

1.4.3　CCStudio 界面操作

操作界面的介绍常常被读者认为是无关紧要的，其实熟悉操作界面对于编程的实际上机工作大有好处。本节将首先介绍整个 CCStudio 的各项菜单，紧接着通过一个编程过程的实例介绍一下如何使用 CCStudio 进行 C/C++程序设计及调试的方法，并向用户展示一个称为“裁缝师”（profiler）的测试程序的工具。在本节中，请读者注意使用 CCStudio 进行程序设计的步骤和具体形式的讲解，编程的语法和具体编程内容我们将在第三章“C/C++程序设计”中介绍。

1．CCStudio 菜单和快捷工具条

CCStudio 的主界面窗口有 12 个菜单，在打开 DSP/BIOS 配置文件时会增加一个 Object 菜单，主界面如图 1-9 所示。窗口的标题栏显示了 Setup 配置信息，窗口中的快捷按钮的意义可以参考帮助“Help”中的内容。将鼠标指针放在相应的按钮上，按 F1 键即可得到相应的帮助。

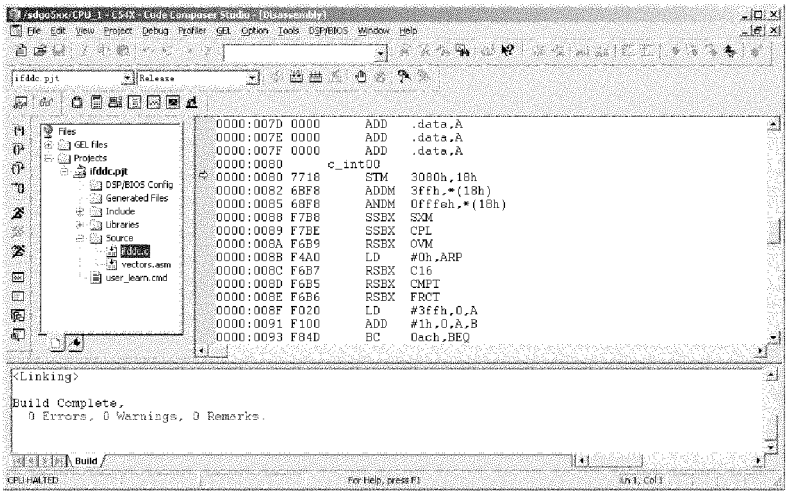


图 1-9　CCStudio 主界面

1) CCStudio 工具条

下面首先介绍 CCStudio 的工具条。

图 1-10 左边为两个下拉列表框，前一个为项目名称，后一个为生成的目标文件的格式，分为 Debug 和 Release 两种。在调试阶段一般使用 Debug，调试成功后应转化为 Release 版写到 FLASH 中去。Release 版中不包括调试信息，为正式发行版本。图 1-10 右边的按钮从左向右依次为编译、汇编连接、全部汇编连接、取消汇编、设置断点、取消断点、设置测试点、取消测试点。

图 1-10　工程工具条

调试工具条如图 1-11 所示，从左向右依次为单步、单步跳过、单步跳出、运行到光标处、运行程序、停止程序、继续运行、观察寄存器、观察存储器、观察堆栈、观察汇编代码。

图 1-11　调试工具条

CCStudio 还包括如图 1-12 所示的变量观测、DSP/BIOS、标准文件操作、编辑、GEL 等工具条，这些工具条给编程和调试带来了方便。Edit 工具条最后一个按钮是指示使用外部编辑器来替代 CCStudio 的编辑器，需要在菜单 Option 中的 Customize/Editor Properties 中设置之后才能使该按钮“使能”。当某些按钮或是菜单呈现浅灰色时，它们不响应鼠标的点击事件，但是可以通过点击另外一些特定的按钮或是菜单，使这些按钮或是菜单“亮”起来，从而可以响应鼠标的点击事件，这个过程称为按钮的“使能”。

图 1-12　其他的工具条

2) CCStudio 主界面

下面再回到图 1-9 中进一步介绍 CCStudio 主界面。在图 1-9 中左边的窗口为工程管理器；右边的为编辑和运行窗口；下面的为编译连接输出信息窗口，在这个窗口中报告程序的出错信息情况，双击编译错误可以直接定位到出错的代码，按 F1 键可以获得更详细的出错原因。

CCStudio 的各项菜单的功用简单介绍如下：

? File：创建源文件、打开一个新的文件、关闭已打开的文件、存储文件、向 DSP 功能板上装入可执行目标程序、工作区的管理(所谓工作区，指包括工程文件在内的一切环境设置，在退出 CCStudio 时，会自动存储一个工作区)、最近打开的文件、退出 CCStudio 等。

? Edit：一些常用的编辑命令、对寄存器、内存和变量等的编辑命令、设置书签等。

? View：显示或关闭主界面窗口下的工具条、对存储器和寄存器的观察、变量观测、混合语言模式等等的观测。

? Project：建立一个工程、打开一个工程、编译一个工程、汇编和连接工程、汇编参数设定、与工程项目管理有关的一些操作。

? Debug：设置断点（必须在可执行语句处）、设置探针、调试程序、复位 DSP、运行程序等等。

? Profiler：“裁缝师”的主要作用是测试程序中函数或是某些区域运行所花费的指令周期数和时间，指出需要重点优化的函数或地方。这个功能对于 C/C++编程尤其重要，几乎所有的 C 程序都必须经过“裁缝师”的验证后才能用于实际 DSP 应用系统中。

? GEL：GEL 对于仿真环境是没有太大用处的，但在模拟环境下，GEL 可以为用户产生一个虚拟的 DSP 硬件初始化环境。

? Option：设置字体、汇编语言样式、存储器映射（在仿真环境下设为不工作或不“使能”），以及用户定制环境（这部分内容比较有意义，读者可以参考在线“帮助”）。

? Tools：Tools 菜单中的选项如 Port Connect、Pin Connect 等是用在模拟环境下的，不能用于仿真环境；而其余的大都只能用于仿真环境，不能用于模拟环境下。其中有一项“Linker Configuration”，将其设为 Visual Linker 时，File/New/Visual Linker Recipe…子菜单项会开启或“使能”，这时工程文件将使用可视化的 Visual Linker 来配置 DSP 应用系统的存储器资源。

? DSP/BIOS：与 DSP/BIOS 工具条中的快捷按钮的功能一一对应。

? Window：与 Borland C++3.1、Visual C++6 的 Window 菜单项的功能一致。

? Help：提供了 CCStudio 的在线帮助和联网帮助。单击 Help 菜单下的 About…子菜单可以看到一个对话框，单击其中的 Component Manager 进入图 1-13 所示的窗口，这个窗口包含了当前使用的 CCStudio 的所有组件及版权信息。

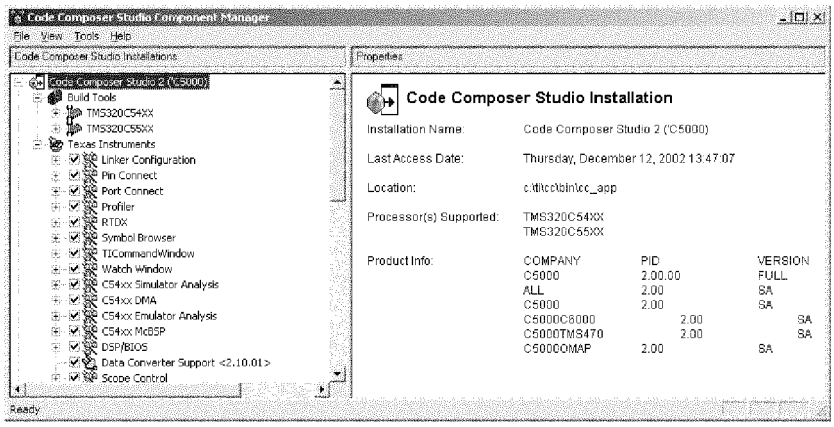


图 1-13　CCStudio 的组件管理器

2．C/C++程序开发步骤

下面我们结合本书建立的实验平台介绍一下 C/C++语言程序的开发步骤（不含 Boot

Loader）。

？ 第一步：给 SY-5402EVM 板和仿真器上电，运行 CCStudio2 进入到 CCStudio 环境下。点击 Project/New，出现如图 1-14 所示的对话框。在 Project 一栏中输入项目名 user_learn（读者可以根据自己的爱好来命名，最好使用见名知义的方法），选择目标处理器为 TMS320C54XX，输出目标文件类型为可执行的.out 文件，文件的默认存储位置为 C:\ti\myprojects\user_learn\目录。点击“完成”出现如图 1-15 所示的窗口。

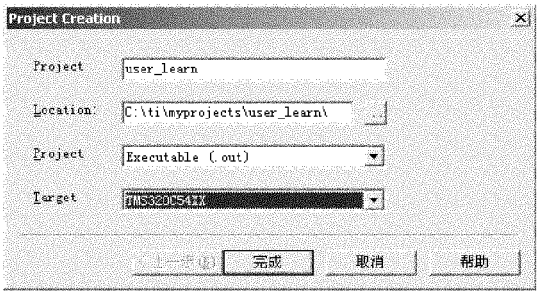


图 1-14　新建工程文件对话框

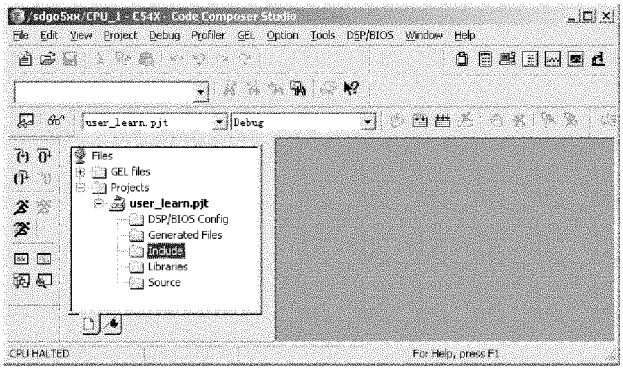


图 1-15　新建工程后的主窗口

？ 第二步：单击 File/New/Source File 编辑一个源文件并命名为 user_learn.c，将它存放到 c:\ti\myprojects\user_learn\目录下。再使用同样的方法编辑 user_learn.cmd、vectors.asm、myinc.h、myface.h 等文件，也都存入到前面提到的目录下。

？ 第三步：在图 1-15 中的工程管理器中，右键单击 user_learn.pjt，在弹出的菜单中选择 Add Files，依次向工程中加入 user_learn.c、user_learn.cmd、vectors.asm、rts.lib、54xdsp.lib 等文件(.h 文件不用添加，在编译时会自动加入)，或在右键单击后的弹出菜单中选择 Scan All Dependencies 子菜单项即可。完成这一步操作之后的主窗口如图 1-16 所示。在用 C/C++语言开发 DSP 程序时，必须借助于工程文件来管理整个工程（或称项目）的所有文件。本例中，user_learn.pjt 文件即为工程文件，它管理了所有包括在工程中的文件。在 CCStudio2 中，可以同时打开多个工程文件，工程文件名为黑体的工程为当前活跃的工程，对于窗口的操作是针对活跃工程文件的操作。多个工程文件一起打开时，只能有一个工程文件处于活跃

状态，可以通过右键单击后的弹出菜单改变当前活跃的工程文件。当只有一个工程文件时，这个工程文件始终处于活跃状态。

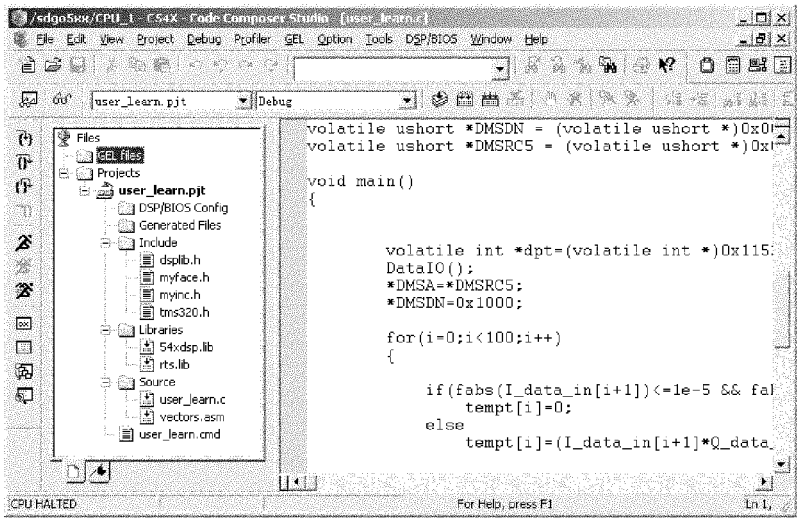


图 1-16　一个完整的 C 语言工程

? 第四步，选择 Project/Rebuild All 对 user_learn.pjt 进行编译连接（编译、汇编和连接使用了同一个命令）。图 1-17 为编译连接的输出信息。编译连接出错后必须到编辑器中进行原程序的修改，再重新编译连接直到没有出错信息为止。这时生成的可执行目标文件 user_learn.out 保存在 c:\ti\myprojects\user_learn\debug 目录下。

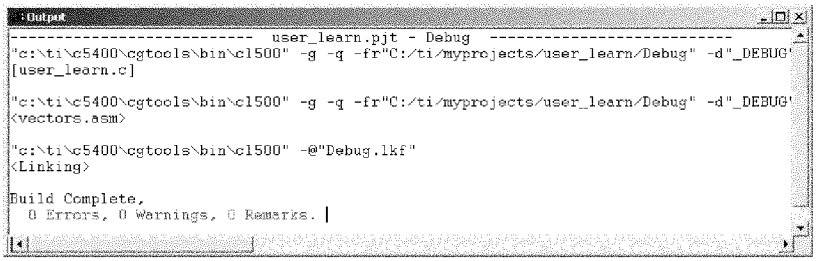


图 1-17　编译连接后的输出信息框

? 第五步，选择 File/Load Program 将目标文件 user_learn.out 装入到 SY-5402EVM 板上的 TMS320VC5402 映射存储器中，单击 Debug/Run 或按 F5 运行程序（有时需要使用 Go main 等对 VC5402 内部的程序指针重定位）。可以借助 View 菜单下的各种观测工具观测运行的中间结果。

? 第六步，观测结果正确后，在 Debug 工具条上将 Debug 改为 Release，再次重新编译连接、运行程序，并将程序转化为 FLASH 固化的格式写入到 FLASH 存储器中（FLASH 存储器是一种可反复擦写、掉电仍能保存数据的 DSP 功能板上常用的程序存储设备，可以通过仿真器和 JTAG 接口在线将程序写入到 FLASH 中）。

3．程序调试

程序调试的通用方法是设置程序断点，程序运行到断点处会自动暂停，这时用户可以使用观测窗口观测中间变量或查看存储器某些相关地址中的数据，进而判定程序运行是否正确，或找出错误出现的大体位置。

可以在中断程序之后使用 CCStudio 打开以下的各种观测窗口，这些窗口的结果分列于图 1-18 至图 1-23。

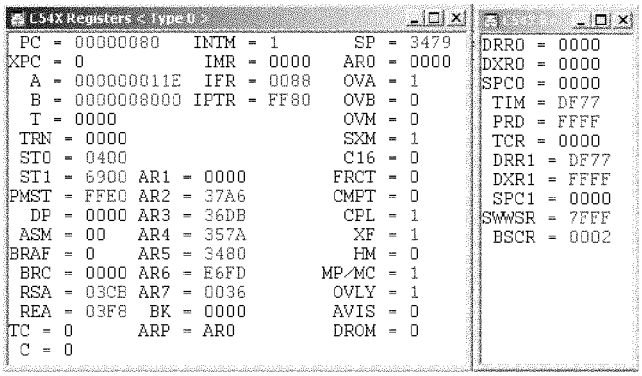


图 1-18　CPU 和外设的映射寄存器

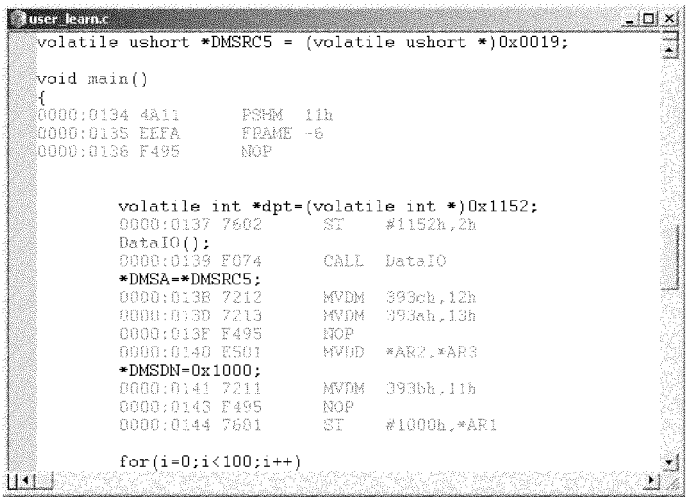


图 1-19　混合语言窗口

通过图 1-18 的观测窗口可以验证 VC5402 初始化是否正确。DSP 芯片的初始化主要是各种寄存器的初始化，初始化不正确，DSP 将无法正常工作。

通过选中 View/Mixed Source/ASM 可以调出图 1-19 所示窗口，这个窗口下可以看到汇编语言和 C/C++语言相比较的直观结果：往往是多条汇编语言共同完成一条 C/C++语句的功能。虽然 C5000 系列 DSP 为定点 DSP 芯片，但是使用 C/C++语言可以编写浮点形式的数据处理语句，编译器自动进行格式转换处理，因此在 CCStudio 环境下，可以使用整型数以

外的其他类型。在图 1-19 窗口下将光标移到某一特定的指令上，按下 F1 键可以直接定位到相应的汇编语言命令解释上，如果读者有兴趣的话，可以仔细分析一下这些汇编语言语句，对一些循环优化的学习会有一定好处。

而图 1-20 是汇编语言的运行窗口，窗口中左边框中的小箭头(屏幕显示为黄色箭头)是程序指针(PC)。复位后，C/C++语言程序执行的入口地址为 c_int00，也是 C/C++语言约定的初始化中断服务程序的入口地址，这个符号定义在 rts.lib 库中。

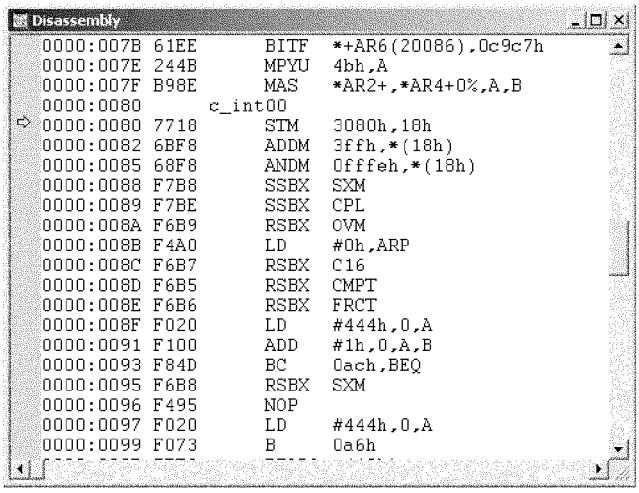


图 1-20　汇编语言代码窗口

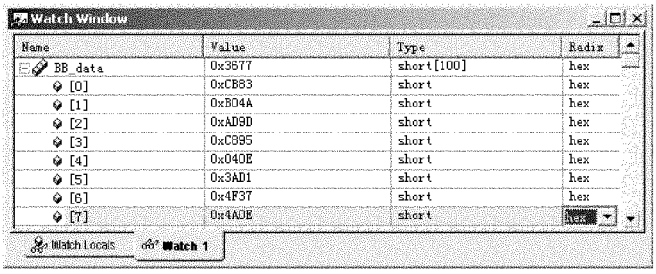


图 1-21　变量观测窗口

本示例程序完成软件无线电常用的正交合成算法。在一帧数据（100 个点）完成运算之后，设置了一个断点，当执行到断点处时，图 1-21 的观测窗口展示了运算结果的数据值（以十六进制形式表示）。这种方法对于数组数据的观测很不直观。

选择 View/Graph/Time/Frequency 子菜单，设置合适的参数后，出现如图 1-22 所示的图形。设置方法请参考图 1-23 所示。

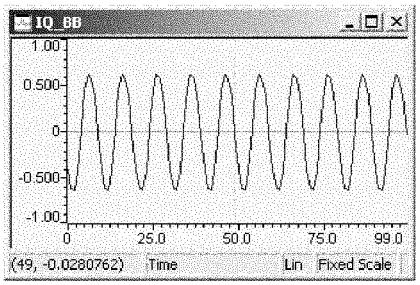


图 1-22　图形观测窗口

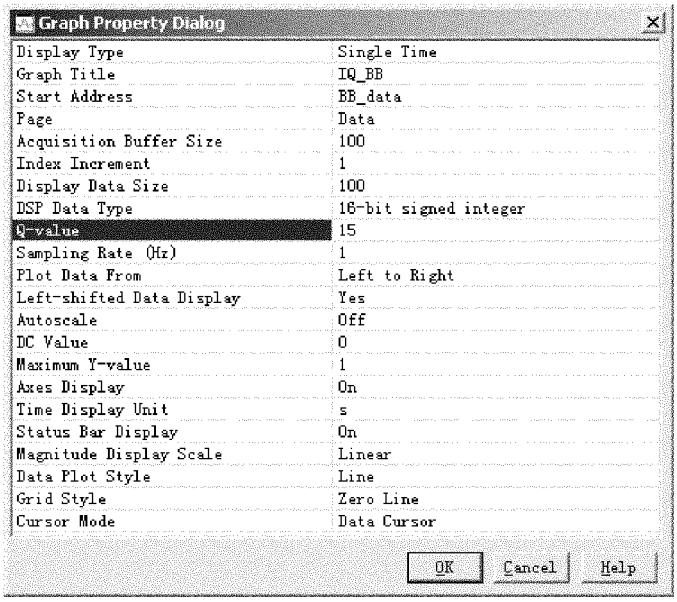


图 1-23　图形观测窗属性设置对话框

对于 VC5402 定点 DSP 来说，一般采用 Q15 的定标方法。Q15 是指将小数点约定在 16 位数据的第 15 位处，定点 DSP 用这种 Qn（n=0,1, …,15）来表示浮点数。图 1-23 中的各项显示了画图的各种属性。其中：Display Type 中指出显示图形的类型，有时间域和频域两种；Start Address 中应填入数据的起始地址；Page 中填入数据所在的页；Acquisition Buffer Size 指获取数据的长度；Display Data Size 指显示数据的长度，一般应小于获取数据的长度才有意义；DSP Data Type 指 DSP 的数据类型；Q Value 指小数点的约定位置。其他选项是画图的线条样式，这里不再说明。

图 1-24 就是著名的“裁缝师”观察窗口，这个窗口中显示了一个示例程序的运行情况。选中 Profiler/Enable Clock，Profiler/Clock Setup 中根据所选用的 DSP 芯片设置指令周期，本例中使用 VC5402，设置 Instruction Cycle 为 10 ns。选中 Profiler/View Clock，选择菜单 Profiler/Start New Session 打开一个新的“裁缝师”，设置 main 函数为“裁缝师”的裁剪函数。这时必须使用 Debug 编译项目，因为使用 Release 编译项目和优化后有些全局符号被重命名而找不到裁剪的函数，而且加入了“裁缝师”后程序的运行会很慢（这是因为“裁缝师”

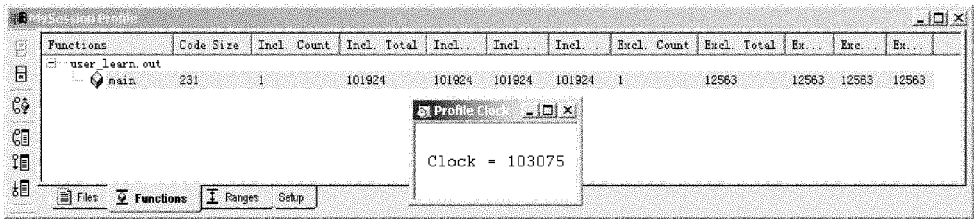


图 1-24 “裁缝师”观测窗口

在准确计算程序的工作量)。从图 1-24 可以看出一帧信号（100 个点）总的运行时钟周期数是 103 075 次，相当于 1 ms 的时间；main 函数的代码共有 231 个字（Code Size=231），该函数调用的函数为 1 个（Incl. Count=1）；包括被调用的函数和该函数一共运行的周期数为 101 924（Incl. Total=101924）；后面两项为最大数和最小数，当没有中断调用时，这三项数值是相等的。仅仅是 main 函数（不包括被调用的函数）运行的周期数为 12 563(Excl. Total=12563)；后面紧跟着的两项也是最大值和最小值。从上面分析来看，仅仅 231 行汇编代码执行了超过 10 万次，这个程序从总体时钟周期数来看几乎不能满足对语音信号的实时处理（采样率为 10 kHz 时，样点间隔为 0.1 ms），从运行的周期数来看更是明显地不合要求。因此，可以得出结论：对语音信号的实时处理是不能按帧信号来完成的。事实上，可以从程序中分析出按点处理时，每个点需要大约 1000 条指令来完成处理，即大约需 1 μs 的时间，所以按点处理是可行的。

从本例的分析可以看出“裁缝师”的重要性，C/C++程序是否高效、可行，通过“裁缝师”的评定后就可以得到准确无误的答案。建议软件工程师和学生在编写 C/C++程序时，重视“裁缝师”的作用，并以“裁缝师”的跟踪结果来决定哪些函数需要进一步改进、优化，哪些函数达到了实时的标准，哪些函数必须重写。不经过“裁缝师”的考验，是不可能编出高效的 C/C++程序的。

1.4.4　GEL 语言

CCStudio 的 GEL 语言是一种交互式的脚本命令，它是解释执行的，即不能被编译成可执行文件。它的作用在于扩展了 CCStudio 的功能，可以用 GEL 来调用一些菜单命令，对 DSP 的存储器进行配置等等。但是作者建议对于使用仿真器和 DSP 功能板的仿真环境用户来说，这种 GEL 语言文件是没有必要加入到配置中的。GEL 语言的重要性在于针对计算机模拟环境的用户，使用 GEL 可以为其准备一个虚拟的 DSP 仿真环境，但也不是非用不可的。

如果用户原来学习过 C++的 Turbo Vision 技术的话，学习 GEL 语言是件非常容易的事。下面给出了 CCStudio 中的 c5402.gel 原程序代码，并作一些说明。(下面程序为源程序引用。)

```
/* set PMST to: MP =    OVLY = 1; DROM off, CLKOUT on */    //GEL 语言采用 C/C++注释法
#define PMST_VAL                0xffe0u
/* set wait-state control reg for: 2 w/s or more on i/o; one for ext memory */
#define SWWSR_VAL               0x2009u
/* set external-banks switch control for: no bank switching; BH set */
#define BSCR_VAL                0x02u           //以上为 CPU 寄存器值的宏定义
/* Set Default Reset Initialization Value */
#define     ZEROS               0x0000u         //零值的宏定义
/* Set Peripheral Control Register Addresses for DEV_RESET */
#define DMPREC                  0x0054u         //下面为 DMA 的地址分配宏定义
#define DMSA                    0x0055u
#define DMSDI                   0x0056u
```

```
#define DMA_CH0_DMFSC_SUB_ADDR   0x0003u
#define DMA_CH1_DMFSC_SUB_ADDR   0x0008u
#define DMA_CH2_DMFSC_SUB_ADDR   0x000Du
#define DMA_CH3_DMFSC_SUB_ADDR   0x0012u
#define DMA_CH4_DMFSC_SUB_ADDR   0x0017u
#define DMA_CH5_DMFSC_SUB_ADDR   0x001cu

#define MCBSP0_SPSA          0x0038u          //下面为串行口外设地址的宏定义
#define MCBSP0_SPSD          0x0039u
#define MCBSP1_SPSA          0x0048u
#define MCBSP1_SPSD          0x0049u
#define MCBSP2_SPSA          0x0034u
#define MCBSP2_SPSD          0x0035u
#define MCBSP_SPCR1_SUB_ADDR      0x0000u
#define MCBSP_SPCR2_SUB_ADDR      0x0001u
#define MCBSP_SRGR1_SUB_ADDR      0x0006u
#define MCBSP_SRGR2_SUB_ADDR      0x0007u
#define MCBSP_MCR1_SUB_ADDR       0x0008u
#define MCBSP_MCR2_SUB_ADDR       0x0009u

#define SRGR1_INIT           0x0001u

#define PRD0                 0x0025u
#define TCR0                 0x0026u

#define PRD1                 0x0031u
#define TCR1                 0x0032u

#define TIMER_STOP           0x0010u
#define TIMER_RESET          0x0020u
#define PRD_DEFAULT          0xFFFFu

#define GPIOCR               0x0010u
//上面的宏定义用于指定寄存器的初始值和寄存器的地址，这种方法在 C/C++编程中常用

/* The Startup() function is executed when the GEL file is loaded. */

StartUp()
{
```

```
    C5402_Init();          //使用这个函数对 C5402 进行初始化，完成后输出下面的字串

    GEL_TextOut("Gel StartUp Complete.\n");        //这是一个 GEL 命令函数
}

menuitem "C5402_Configuration";        //这是一个 GEL 菜单，如图 1-25 所示
hotmenu CPU_Reset()
{
    GEL_Reset(); //CPU_Reset 热键菜单及其调用的函数
    PMST = PMST_VAL;
```

图 1-25　GEL 菜单

```
/* don't change the wait states, let the application code handle it */
/* note: at power up all wait states will be the maximum (7)     */
/*     SWWSR = SWWSR_VAL;                                        */

    BSCR = BSCR_VAL;              //对 SWWSR 和 BSCR 控制寄存器进行初始化

    GEL_TextOut("CPU Reset Complete.\n");
}

/* All memory maps are based on the PMST value of 0xFFE0 */

hotmenu C5402_Init()        //C5402_Init 热键菜单及其调用函数
{
    GEL_Reset();
    PMST = PMST_VAL;

/* don't change the wait states, let the application code handle it*/
/* note: at power up all wait states will be the maximum (7)    */
/*     SWWSR = SWWSR_VAL;                                       */
    BSCR = BSCR_VAL;
    C5402_Periph_Reset();
    GEL_XMDef(0,0x1eu,1,0x8000u,0x7fu);    //下面为 GEL 的存储器分配及映射
    GEL_XMOn();
    GEL_MapOn();
    GEL_MapReset();

    GEL_MapAdd(0x80u,0,0x3F80u,1,1);        /* DARAM */
    GEL_MapAdd(0x4000u,0,0xC000u,1,1);      /* External */
```

```
    GEL_MapAdd(0x18000u,0,0x8000u,1,1);    /* Extended Addressing - Page 0 */

    GEL_MapAdd(0x0u,1,0x60u,1,1);          /* MMRs */
    GEL_MapAdd(0x60u,1,0x3FA0u,1,1);       /* DARAM */
    GEL_MapAdd(0x4000u,1,0xC000u,1,1);     /* External */

    GEL_TextOut("C5402_Init Complete.\n");
}
/* ****************************** */

C5402_Periph_Reset()                   //调用子函数进行外设初始化
{
    IFR = 0xFFFF;
    IFR = 0x0000;

    DMA_Reset();                       //DMA 初始化函数调用入口
    MCBSP0_Reset();                    //串口 0 初始化函数调用入口
    MCBSP1_Reset();                    //串口 1 初始化函数调用入口
    TIMER0_Reset();                    //计时器 0 初始化函数调用入口
    TIMER1_Reset();                    //计时器 1 初始化函数调用入口
    GPIO_Reset();                      //通用 IO 初始化函数调用入口
}

DMA_Reset()                  //DMA 初始化函数
{
    *(int *)DMPREC = ZEROS;

    *(int *)DMSA = DMA_CH0_DMFSC_SUB_ADDR;
    *(int *)DMSDI = ZEROS;
    *(int *)DMSDI = ZEROS;
    *(int *)DMSA  = DMA_CH1_DMFSC_SUB_ADDR;
    *(int *)DMSDI = ZEROS;
    *(int *)DMSDI = ZEROS;

    *(int *)DMSA  = DMA_CH2_DMFSC_SUB_ADDR;
    *(int *)DMSDI = ZEROS;
    *(int *)DMSDI = ZEROS;

    *(int *)DMSA  = DMA_CH3_DMFSC_SUB_ADDR;
```

```
    *(int *)DMSDI = ZEROS;
    *(int *)DMSDI = ZEROS;

    *(int *)DMSA   = DMA_CH4_DMFSC_SUB_ADDR;
    *(int *)DMSDI = ZEROS;
    *(int *)DMSDI = ZEROS;

    *(int *)DMSA   = DMA_CH2_DMFSC_SUB_ADDR;
    *(int *)DMSDI = ZEROS;
    *(int *)DMSDI = ZEROS;
}

MCBSP0_Reset()                  //串行口 0 的初始化函数
{
    *(int *)MCBSP0_SPSA = MCBSP_SPCR1_SUB_ADDR;
    *(int *)MCBSP0_SPSD = ZEROS;
    *(int *)MCBSP0_SPSA = MCBSP_SPCR2_SUB_ADDR;
    *(int *)MCBSP0_SPSD = ZEROS;

    *(int *)MCBSP0_SPSA = MCBSP_SRGR1_SUB_ADDR;
    *(int *)MCBSP0_SPSD = SRGR1_INIT;
    *(int *)MCBSP0_SPSA = MCBSP_SRGR2_SUB_ADDR;
    *(int *)MCBSP0_SPSD = ZEROS;

    *(int *)MCBSP0_SPSA = MCBSP_MCR1_SUB_ADDR;
    *(int *)MCBSP0_SPSD = ZEROS;
    *(int *)MCBSP0_SPSA = MCBSP_MCR2_SUB_ADDR;
    *(int *)MCBSP0_SPSD = ZEROS;
}

MCBSP1_Reset()                  //串行口 1 的初始化函数
{
    *(int *)MCBSP1_SPSA = MCBSP_SPCR1_SUB_ADDR;
    *(int *)MCBSP1_SPSD = ZEROS;
    *(int *)MCBSP1_SPSA = MCBSP_SPCR2_SUB_ADDR;
    *(int *)MCBSP1_SPSD = ZEROS;

    *(int *)MCBSP1_SPSA = MCBSP_SRGR1_SUB_ADDR;
    *(int *)MCBSP1_SPSD = SRGR1_INIT;
```

```
    *(int *)MCBSP1_SPSA = MCBSP_SRGR2_SUB_ADDR;
    *(int *)MCBSP1_SPSD = ZEROS;

    *(int *)MCBSP1_SPSA = MCBSP_MCR1_SUB_ADDR;
    *(int *)MCBSP1_SPSD = ZEROS;
    *(int *)MCBSP1_SPSA = MCBSP_MCR2_SUB_ADDR;
    *(int *)MCBSP1_SPSD = ZEROS;
}


TIMER0_Reset()      //计时器 0 的初始化
{
    *(int *)TCR0 = TIMER_STOP;
    *(int *)PRD0 = PRD_DEFAULT;
    *(int *)TCR0 = TIMER_RESET;
}

TIMER1_Reset()          //计时器 1 的初始化
{

    *(int *)TCR1 = TIMER_STOP;
    *(int *)PRD1 = PRD_DEFAULT;
    *(int *)TCR1 = TIMER_RESET;
}

GPIO_Reset()            //通用 IO 中的初始化
{
    *(int *)GPIOCR = ZEROS;
}
//c5402.gel 文件结束
```

将 GEL 文件列于此处的原因在于：第一，在实际 DSP 程序设计时，这些初始化工作必须有相应的 C/C++语句来完成，C/C++程序执行时必须首先做的工作就是这些硬件的初始化工作，这些 GEL 语言可以提供一个参考；第二，GEL 语言的风格与实际 C/C++语言是完全一样的，有些语句是可以完全照搬的，这对 C/C++语句的初始化是一个不小的可走捷径；第三，因为 GEL 语言是一种解释命令，也有一些语句是不能用于下载到 DSP 中的 C/C++语言程序中的，也就是说，有些 GEL 语句离开了 CCStudio 环境就没有效用了，因此要注意区分。

下面列出 GEL 的函数，这有助于读者对上面程序的进一步分析。

GEL_Animate：开始执行程序；
GEL_BreakPtAdd：加入新的断点；
GEL_BreakPtDel：删除一个存在的断点；
GEL_BreakPtReset：删除所有断点；
GEL_CloseWindow：关闭一个存在的输出窗口（指计算机上 CCStudio 的 Out 窗口）；
GEL_Exit：关闭活动的控制窗口；
GEL_Go：运行目标程序至特定的语句；
GEL_Halt：中断目标程序；
GEL_Load：装入一个 COFF 文件（COFF 文件指目标文件的格式为 COFF 格式）；
GEL_MapAdd：添加一块映射存储器空间；
GEL_MapDelete：删除一块映射存储器空间；
GEL_MapOn：存储器映射启动或“使能”；
GEL_MapDelete：存储器映射不启动或不“使能”；
GEL_MapReset：清空存储器映射；
GEL_MemoryFill：以特定的值填充一块映射存储区；
GEL_MemoryLoad：从计算机硬盘上的一个文件中装入数据到存储区中；
GEL_MemorySave：将存储区中的数据保存到硬盘中的一个文件中；
GEL_OpenWindow：打开一个输出窗口；
GEL_PatchAssembly：用特定的指令填充存储区小块；
GEL_ProjectBuild：编译、汇编和连接当前工程；
GEL_ProjectLoad：装入工程到目标板上；
GEL_ProjectRebuildAll：重新编译连接当前工程；
GEL_Reset：复位目标板；
GEL_Restart：复位程序指针到程序入口点；
GEL_Run：开始运行目标程序；
GEL_RunF：开始运行目标程序，并断开与目标板的连接；
GEL_SymbolLoad：仅装入标号到目标板；
GEL_System：执行 DOS 命令；
GEL_TargetTextOut：输出格式化的目标字符串；
GEL_TextOut：输出文本到输出窗口中；
GEL_WatchAdd：添加表达式到观测窗口中；
GEL_WatchDel：删除观测窗口中的表达式；
GEL_WatchReset：删除观测窗口中的所有表达式；
GEL_XMDef：定义扩展的存储区范围；
GEL_XMOn：启动或“使能”扩展的映射存储区。

上述各命令的用法可以在帮助中查看，也可以将光标移动到命令上按 F1 键即可。

1.4.5　Visual Linker 操作方法

Visual Linker 提供了可视化配置 DSP 存储器的方法，是典型的“所建即所得”的操作工具。对于资源比较紧张的用户，可以使用这种方法直观地观察存储占用情况。作者建议初学者不要采用这种方法。按照第 1.4.3 节的方法装入程序，但不要装入 user_learn.cmd 文件（装入了也可以，但是系统会提示用户修改为 .rcp 格式的文件，用户可以自行试一下），打开 Tools/Linker Configuration 的窗口，如图 1-26 所示，选中 Use the Visual Linker 单选钮。Use the text linker 按钮的作用是使用文本的 .cmd 文件形式。Visual Linker 的输入为汇编后的目标文件，Visual Linker 会显示所有目标文件中的程序、数据和标号，用户使用拖拉的方法将这些部分分配到相应的映射存储区中。

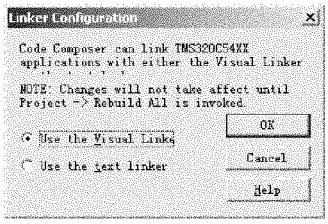


图 1-26　Linker 配置

这时编译连接这个工程，会出现如图 1-27 所示的错误，双击这个错误码会出现 Visual Linker 处方向导（Visual Linker 生成的文件称为处方或 linker 处方），选择一个处方模板，最后出现如图 1-28 所示的 Visual Linker 窗口。

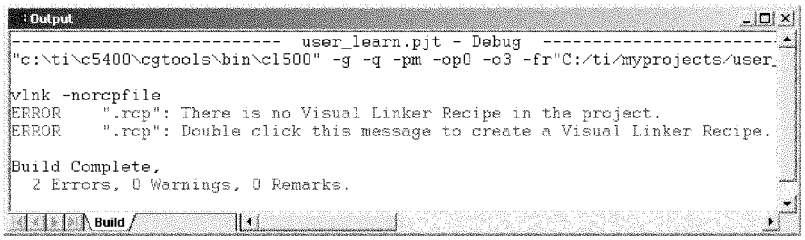


图 1-27　没有加入处方时的编译错误

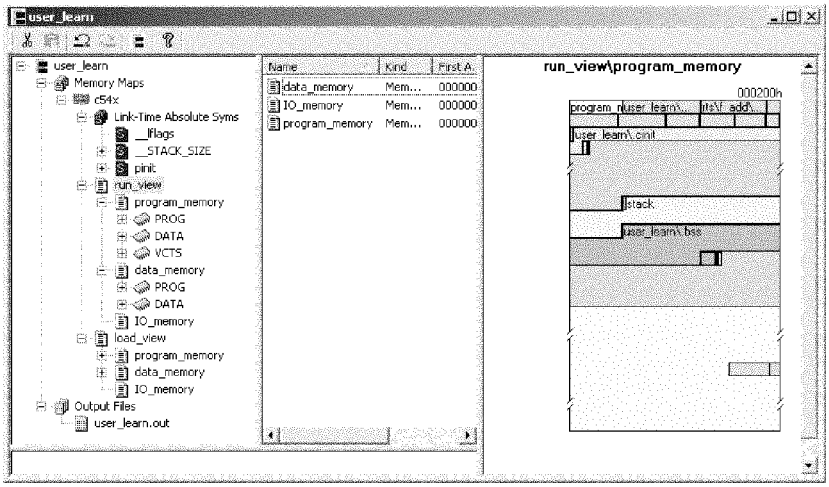


图 1-28　处方的设置

对于从模板创建来的一个新的处方.rcp，某些段会出现“Not Yet Placed”，表示这些段还没有被分配。应将鼠标放在一个类似于文件夹的地方，按住不放将其拖拉到图 1-28 右边的区域中，这个区域可以通过鼠标右键选择为 DATA 区还是 PROGRAM 区，甚至可以选择 I/O 空间，但没有多大意义。当用鼠标右键点击相应的段时，可以通过弹出菜单进入其属性区，设置这个段是否可读、可写、可装入执行程序、可装入初始化数据等重要属性。图 1-28 为一个创建好的 user_learn.rcp 处方。

打开 Visual Linker 的另一个方法，是通过菜单 File/New/Visual Linker Recipe 来打开。当然也可以直接从硬盘中找到 Visual Linker 执行文件，进入 Visual Linker 环境。Visual Linker 和 .cmd 格式的文件作用没有本质的区别，但是 Visual Linker 更直观一些，对用户的要求特别是对用户对 DSP 存储器的了解程序要求更高一些。所以，建议初学者使用 .cmd 格式的文本方式。

1.5 本 章 小 结

本章详细介绍了学习和开发 DSP 应用系统所必备的硬件设备和软件平台，介绍了设计一个程序实例的方法，并说明了调试面向 DSP 的 C/C++程序的方法。在本章中也介绍了 GEL 语言和 Visual Linker 编辑器的使用方法，特别强调了“裁缝师”的重大用途。

本章开始提出的三个问题在本章中均有详细的解答，在此仅补充一点关于 C/C++语言的执行时间的评定方法。对于一个面向 DSP 的 C/C++程序，最重要的一点在于其实时性和高效性。因此，必须对 C/C++程序的执行情况有一个完整的了解。在这方面，CCStudio 提供了“裁缝师”（Profiler）技术，可以定量化地监测程序中的函数、程序范围等等的执行情况，并报告了详细的运行周期数和函数调用情况等等。没有“裁缝师”将很难想象 C/C++语言编程的前景。现在有了“裁缝师”，软件设计人员可以通过“裁缝师”，把工作量集中在那些“裁缝师”认为是不合理的程序段或函数上（对于 C/C++来说，主要是函数），不断地改进算法直到达到要求为止。通过本章的学习，应充分地掌握“裁缝师”的应用方法，并应用在实际 C/C++程序设计中。考虑到编译连接成可执行文件的紧凑性和高效性，应尽可能地在程序调试成功后，将函数改为内联函数（即用 inline 关键字开头）。

最后补充一点关于编程风格方面的知识：一个程序设计者应合理地利用空格缩进和换行来控制程序的不同功能语句组的相对位置，养成一种习惯和风格。这对本书的学习也是有帮助的。

习　题　一

1. 什么是 DSP?
2. DSP 的用途是什么？
3. 开发 DSP 应用系统的过程应怎样进行？
4. 使用 C/C++设计 DSP 程序比用汇编语言效率低(高)多少？
5. 什么是 DSP 的仿真环境和模拟环境？
6. 仿真环境下设计一个 C/C++程序的过程是怎样进行的？

7．为什么模拟环境下的程序调试会比仿真环境下慢得多？
8．CCStudio 中能不能装入多个 GEL 文件？
9．C5000 定点 DSP 支持浮点的 C/C++程序设计吗？
10．“裁缝师”的作用是什么？
11．如何在 CCStudio 中使用断点和探针？
12．如何设置图形观测窗口的属性？
13．如何使用 Visual Linker 进行存储区域的映射配置？
14．如何设置一个程序的编译选项卡？
15．程序编译生成的 Debug 版和 Release 版有什么区别？
16．如何在 Windows 2000+SP3 环境下安装 DSP 仿真器？

第二章

TMS320C5000系列硬件基础

2.1 本章内容简介

本章内容分为三部分。前两部分讲解C5000系列DSP芯片的特性和内部结构(C5000系列包括C5x、C54x和C55x三种型号)。为了避免泛泛而谈，本章选取了具有代表性的TMS320VC5402芯片和TMS320VC5510芯片，以此为例，仅就与C/C++编程有关的CPU、存储器配置、片上外设和中断做了全面分析。第三部分对三意电子有限公司的SY-5402EVM板作了详细介绍，针对SY-5402EVM板给出了存储配置文件和中断向量表，后续的程序均在该SY-5402EVM板上调试通过。希望这些工作可以起到解剖一只麻雀的作用。

学习这部分内容对学习C/C++语言编程可以起到间接的推动作用，可以使读者熟悉C5000系列的资源分配利用情况，为下一步学习DSP的C语言程序设计在硬件资源使用上打下基础。本章给出的存储配置文件（.cmd）和中断向量表文件（vectors.asm）为第三章中的程序设计作准备。

思考题

(1) 如何对TMS320VC5410的存储器进行配置？

(2) 如何访问各种片上物理存储器？

(3) 如何编写C/C++语言程序的中断向量表？

2.2 TMS320VC5402 简介

2.2.1 CPU

TMS320C54x 系列 DSP 有相同的 CPU 结构。

TMS320VC5402（以下简称 VC5402）的 CPU 结构特征如下：

(1) 具有高性能的改进的哈佛总线结构，即具有三条独立的 16 bit 数据存储器总线和一条 16 bit 的程序存储器总线；

(2) 具有一个 40 bit 的算术逻辑单元，包括一个 40 bit 的筒形移位器和两个独立的加法器；

(3) 17×17 bit 的并行乘法器与专用的 40 bit 加法器相结合可以在一个非并行指令周期内完成一次乘加操作（MAC）；

(4) 具有专用于 Viterbi 蝶形算法的比较、选择和存储单元（CSSU）；

(5) 指数译码器可以在一个指令周期内求一个 40 bit 累加数的指数值，这里的指数定义为累加器中没有数据占用的位数的个数减去 8，因此，指数的范围为 -8～31。

(6) 两个地址发生器、八个辅助寄存器和两个辅助寄存器算术单元（ARAU）；

(7) 单周期定点指令执行时间为 10 ns。

2.2.2 存储器

TMS320C54x 系列 DSP 内部均带有一定数量的高速物理存储区空间，在实时性要求很严格的应用系统中，应尽量将程序和数据存放在内部物理存储区中，而且尽可能地将数据区定义在内部双访问 RAM（DARAM）中，程序区可定义在内部单访问 RAM（SARAM）、DARAM 或是 ROM 中，一些查找表或是初始化数据也可以放在程序区中。因为对于程序区常常只有读操作，而对于数据区往往可以同时存在有读操作和写操作，所以数据区尽可能定义在 DARAM 中。对片内物理存储器的访问是通过访问映射存储器来实现的，也就是说，片内物理存储器必须被映射到映射存储器上才能被访问。DSP 系统的映射存储器分为三块区域，分别称为程序区、数据区和 IO 区。一般来说，IO 区是片外资源，访问空间大小为 64 K×16 bit；数据区可以为片上存储区映射的，也可以是片外存储器映射的，或兼而有之，访问空间大小也是 64 K×16 bit，而且，这两个区域常常是不能被扩展访问的。程序区分为基本程序区和扩展程序区，显然是可以被扩展的。基本程序区的访问空间也是 64 K×16 bit，对于不同的 DSP 芯片，扩展能力不同。对于 VC5402 来说，最大扩展访问空间为 1024 K×16 bit。可见，DSP 系统的映射存储器代表了 DSP 芯片的一种寻址能力和可访问空间的大小，在没有对映射存储器配置前，这些映射存储空间是虚拟的，是不能用来存储程序和数据的。所以在 DSP 程序的编译和汇编之后，连接成目标文件之前，必须加入存储器配置文件（.cmd），将实际的物理存储区空映射到映射存储器空间上。

VC5402 仅提供了 4 K×16 bit 的片上 ROM 和 16 K×16 bit 的片上 DARAM。DARAM 由 2 块 8 K×16 bit 的区块组成，每一个区块均可在一个指令周期内完成两次读操作或一次读和一次写操作。映射存储器的配置受到 VC5402 外部管脚 MP/MC 以及处理器模式状态寄

存器 PMST 的控制。具体配置方案如图 2-1 所示。常采用的方案是 MP/MC=0 且 OVLY=1 的情况，其中 OVLY 为 16 bit 寄存器 PMST 的第 5 位。当 VC5402 上电硬复位且 MP/MC=0 时，片上物理 ROM 会映射到程序映射存储器的 FF00h 至 FFFFh 地址范围内，芯片会自动调用片上 ROM 中固化的 Boot Loader 程序。关于 Boot Loader 的进一步讨论请参考第六章“Boot Loader 程序设计”。在 VC5402 硬件复位成功后，可以通过编程的方法设置 PMST 寄存器中 IPTR 的值，重新定位中断向量表在程序区中的位置，IPTR 占据了 PMST 的第[15:7]位。PMST 寄存器如图 2-2 所示。IPTR 重新定位中断向量表位置后，用户设置的中断向量表将被映射到这个新的位置上。中断向量表固定地占有 128×16 bit 的区域，所以重新定位后的中断向量表应该占有 IPTR 的二进制数值后面补上 7 个 0 作为开始存储位置以及之后的 128×16 bit 区域。图 2-1 和图 2-2 均来自 TMS320VC5402 芯片资料，在这里只是引用来说明以上的解释内容。

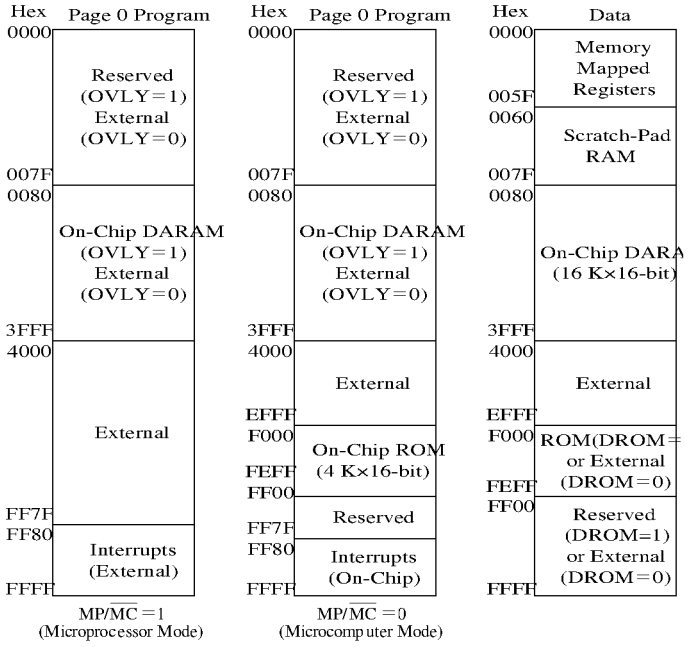


图 2-1　VC5402 映射存储器结构

注：本图原文给出，若译成中文则显得不伦不类。但为方便理解，下面给出中文解释：

Page 0 Program　程序区的第 0 页

Data　数据区

Hex　以十六进制数表示

Reserved(OVLY=1) External(OVLY=0)　当 OVLY 设为 1 时，本区段保留；当 OVLY 设为 0 时，本区段映射到外部存储空间

On-Chip DARAM(OVLY=1) External(OVLY=0)　当 OVLY 设为 1 时，本区段被片上双访问 RAM 占据；当 OVLY 设为 0 时，本区段映射到外部存储空间

External　　本区段映射到外部存储空间

Interrupts(External)　中断向量表存储区，存储在外部存储空间

Microprocessor Mode　微处理器工作模式

Microcomputer Mode　微机工作模式

On-Chip ROM(4K*16 bit)　本区段被片上 4KW 的 ROM 占据

Reserved　保留

Interrupts(On-Chip)　中断向量表存储区，存储在片上存储空间内

Memory Mapped Registers　内存映射寄存器存储区

Scratch-Pad RAM　临时 RAM 区

On-Chip DARAM(16K*16 bit)　本区段被片上双访问 RAM 占据

ROM（DROM=1）or External(DROM=0)　当 DROM=1 时，本区段映射到 ROM 上；当 DROM=0 时，本区段映射到外部存储空间

Reserved(DROM=1)or External(DROM=0)　当 DROM=1 时，本区段保留；当 DROM=0 时，本区段映射到外部存储空间

15 ~ 7	6	5	4	3	2	1	0
IPTR	MP/$\overline{MC}$	OVLY	AVIS	DROM	CLK OFF	SMUL	SST
R/W	R/W	R/W	R	R	R	R/W	R/W

LEGEND: R=Read,W=Write(R 代表寄存器中的某位是只读的，W 代表某位是可写的，R/W 代表某位是可读写的。)

图 2-2　PMST 处理器模式状态寄存器

例如，当 VC5402 硬复位后，通过编程方法设置 PMST 为 0x00A0，则中断向量表会重新定位到 0x80 至 0xFF 的区间内，因此这个区间不能再分配给其它程序或是数据使用。

由图 2-1 可以看出，当设置 MP/MC=0，OVLY=1 时，实际能被访问的存储区域是很有限的，VC5402 的 ROM 区基本上是保留给 DSP 芯片本身的，DARAM 区同时被映射到程序映射存储区（PROGRAM 区）和数据映射存储区（DATA 区）上相同的地址范围上。因此 VC5402 的两个性质不同的映射存储区同时占有同一块物理存储区，这时，对 VC5402 进行存储空间分配时，已经分配给 PROGRAM 区的地址范围就不能再分配给 DATA 区使用，同样道理，分配给 DATA 区的地址范围也不能再分配给 PROGRAM 区。一个详细的存储区配置文件（.cmd）请参考第 2.4 节“SY-5402EVM 板”。

VC5402 具有 20 条地址线，映射程序空间最大为 16 页×(64 K×16 bit/页)，即 1024 K×16 bit 的空间大小。扩展后的程序映射存储空间如图 2-3 所示(该图来自 VC5402 芯片资料)。当 OVLY=1 时，第 1 页至第 15 页的低端 16 K×16 bit 是映射到第 0 页上的相应地址处；当 OVLY=0 时，这些页是独立存在的。

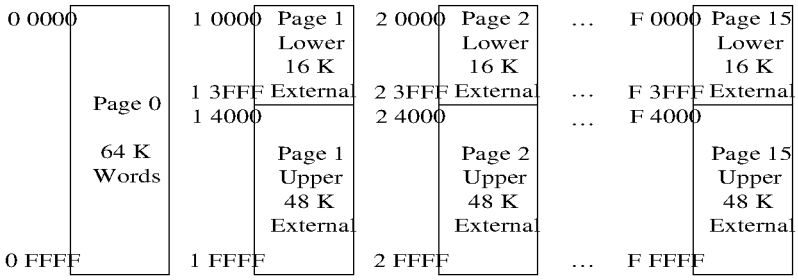


图 2-3　VC5402 扩展的程序存储器

Page 0 64K Words　第 0 页 空间大小为 64 KW

Page 1 Lower 16K External　第 1 页到第 15 页的低端 16 KW 空间映射到外部存储空间上

Page 2 Lower 16K External　第 2 页的低端 16 KW 空间映射到外部存储空间上

⋮

Page 1 Upper 48K External　第 1 页的高端 48 KW 空间映射到外部存储空间上

Page 2 Upper 48K External　第 2 页的高端 48 KW 空间映射到外部存储空间上

2.2.3　片上外设

VC5402 具有以下片上外设：

(1) 具有软件可编程的等待状态发生器、可编程的块切换等待状态；

(2) 具有并行 I/O 口；

(3) 具有一个增强的 8 bit 主机接口(HPI8)；

(4) 具有两个多通道缓冲串行口(McBSPs)；

(5) 具有两个硬件计时器；

(6) 具有一个带有锁相环(PLL)的时钟发生器；

(7) 具有一个直接内存访问(DMA)控制器。

下面对上述每一种片上外设作详细介绍。

1．软件可编程的等待状态发生器

当 VC5402 访问片外低速资源时，可以在访问周期内最多插入 14 个等待时钟周期。如果仍不能满足要求时，应将外部低速资源的读写控制线与 VC5402 的 READY 线相连接，访问由 READY 来控制。当片外资源速度足够高时，可以设置等待状态发生器不工作从而进一步节约功耗。等待状态发生器是一种软设备，完全由软件等待状态寄存器（SWWSR）来控制。SWWSR 的低 14 bit 按每 3 bit 一组分成 5 个独立的地址范围，如图 2-4 所示。每个地址范围针对不同的外部资源分别可以设置 0 到 7 个等待状态。软件等待状态控制寄存器（SWCR）的软件等待状态乘数位（SWSM）定义等待状态数的倍数是 1 还是 2，如图 2-5 所示。VC5402 上电硬复位后，等待状态发生器初如化为对所有外部资源访问等待 7 个状态数。表 2-1 和表 2-2 解释了 SWWSR 和 SWCR 两个寄存器。这几个图表都是来自 VC5402 资料，文中仅是引用来解释一下。

15	14 12	11 9	8 6	5 3	2 0
XPA	I/O	Data	Data	Program	Program
R/W-0	R/W-111	R/W-111	R/W-111	R/W-111	R/W-111

LEGEND: R=Read,W=Write,0=Value after reset(R 代表某位只读，W 代表某位可写，R/W 代表某位可读可写，-0 代表芯片复位后的初始值为 0，-111 代表芯片复位后的初始值为 111。)

图 2-4　SWWSR 寄存器

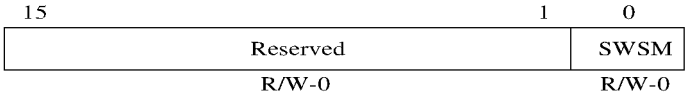
表 2-1　SWWSR 寄存器中各位的含义

BIT		RESET VALUE (复位值)	FUNCTION （功能）
NO.	NAME		
15	XPA	0	Exlended program address control bit. XPA is used in conjundion with the program space fields (bits 0 through 5) to select the address range for program space wait states 外部程序地址控制位（eXtended Program Address），XPA 与程序空间位域（Program：本寄存器的第 0 位到第 5 位）联合使用，为程序空间等待状态选择地址范围
14-12	I/O	1	I/O space. The field value (0-7) corresponds to the base number of wait states for I/O space accesses within addresses 0000-FFFFh. The SWSM bit of the SWCR defines a multiplication factor of 1 or 2 for the base number of wait states I/O 空间标志位域。这个位域可以设置为 0 至 7，即设置 I/O 空间的 0x0000 至 0xFFFF 地址范围的访问等待时钟周期数为 0 至 7。SWCR 寄存器中的 SWSM 位为 1 时，实际等待时钟数为这个位域的值乘以 2；SWCR 寄存器中的 SWSM 位为 0 时，等待时钟数保持不变
11-9	Data	1	Upper data space. The field value (0-7) corresponds to the base number of wait states for extemal data space accesses within addresses 8000-FFFh. The SWSM bit of the SWCR defines a multiplication factor of 1 or 2 for the base number of wait states. 高端数据空间。这个位域的值可以设置为 0 至 7，对应于访问外部数据空间 0x8000 至 0xFFFF 地址范围的基本等待时钟周期数。SWCR 寄存器中的 SWSM 位为 1 时，实际等待时钟数为这个位域的值乘以 2；SWCR 寄存器中的 SWSM 位为 0 时，等待时钟数保持不变
8-6	Data	1	Lower data space. The field value (0-7) corresponds to the base number of wait states for extemal data space accesses within addresses 0000-7FFFh. The SWSM bit of the SWCR defines a multiplication factor of 1 or 2 for the base number of wait states 低端数据空间。这个位域的值可以设置为 0 至 7，对应于访问外部数据空间 0x0000 至 0x7FFF 地址范围的基本等待时钟周期数。SWCR 寄存器中的 SWSM 位为 1 时，实际等待时钟数为这个位域的值乘以 2；SWCR 寄存器中的 SWSM 位为 0 时，等待时钟数保持不变

续表

BIT		RESET VALUE (复位值)	FUNCTION （功能）
NO.	NAME		
5-3	Program	1	Upper program space. The field value (0-7) corresponds to the base number of wait states for extemal program space accesses within the following addresses: □ XPA=0: x8000-xFFFh □ XPA=1: The upper program space bit field has no effect on wait states. The SWSM bit of the SWCR defines a multiplication factor of 1 or 2 for the base number of wait states 高端程序空间标志位域。这个位域的值可以设置为 0 至 7，对应于访问外部程序空间的基本等待时钟数。当 XPA=0 时，地址范围为 0x8000 至 0xFFFF；当 XPA=1 时，此位域被屏蔽掉。SWCR 寄存器中的 SWSM 位为 1 时，实际等待时钟数为这个位域的值乘以 2；SWCR 寄存器中的 SWSM 位为 0 时，等待时钟数保持不变
2-0	Program	1	Program space. The field value (0-7) corresponds to the base number of wait states for extemal program space accesses within the following addresses: □ XPA=0: x0000-x7FFFh □ XPA=1: 00000-FFFFFh The SWSM bit of the SWCR delines a multiplication factor of 1 or 2 for the base number of wait states 程序空间标志位域。这个位域的值可以设置为 0 至 7，对应于访问外部程序空间的基本等待时钟数。当 XPA=0 时，地址范围为 0x0000 至 0x7FFF，当 XPA=1 时，地址范围为 0x00000 至 0xFFFFF。SWCR 寄存器中的 SWSM 位为 1 时，实际等待时钟数为这个位域的值乘以 2；SWCR 寄存器中的 SWSM 位为 0 时，等待时钟数保持不变

图 2-5 中的－0 和图 2-4 中的－0，…，－111 等意义相同，指在上电复位时初始化值。



LEGEND：R=Read,W=Write(R 表示可读，W 表示可写，R/W 表示可读可写)

图 2-5　SWSR 寄存器

表 2-2　SWSR 寄存器各位的意义

PIT		RESET VALUE (复位值)	FUNCTION （功能）
NO.	NAME		
15-1	Reserved	0	These bits are reserved and are unaffected by writes 这些位被芯片保留，对这些位域的写操作将被忽略（或称视为无效写操作）
0	SWSM	0	Software wait-stale multiplier. Used to multiply the number of wait states defined in the SWWSR by a factor of 1 or 2 □SWSM=0:wait-state base values are unchanged(multiplied by 1). □SWSM=1:wait-state base values are multiplied by 2 for a maximum of 14 wait states. 软件等待状态倍乘位（Software Wait-State Multiplier），用于指定在 SWWSR 寄存器中定义的基本等待状态数的倍数。当 SWSM=0 时，基本等待状态数保持不变；当 SWSM=1 时，基本等待状态数乘以 2，此时等待状态数的范围为 0 至 14

备注：作者直接引用原文的图表，是因为对这些图表国内还没有统一的标准的翻译，而且如果翻译不准确还会引起歧义，读者可以结合文中的介绍很容易地看懂这些图表。在实际开发项目时，参阅相应的 DSP 器件资料是必不可少的，因而在实际开发项目时，应首先熟悉怎么去看器件资料，逐渐摸索出一种分析器件资料的方法。如果这些图表不清楚，可以利用在线“帮助”，输入名称去索引相关内容。

2．可编程的块切换等待状态

VC5402 具有块切换逻辑功能。当访问在 VC5402 的同一存储区内的不同区块切间换时，或者由访问数据区转向访问程序区时，VC5402 会自动插入一个时钟周期来避免总线竞争。块切换控制寄存器（BSCR）定义了块切换等待状态的块大小。BSCR 寄存器如图 2-6 所示，其各位的意义如表 2-3 所示。

15　12	11	10　3	2	1	0
BNKCMI	PS-DS	Reserved	HBH	BH	EXIO
R/W-1111	R/W-1	R-0	R/W-0	R/W-0	R/W-0

LEGEND：R=Read,W=Write(R 表示可读，W 表示可写，R/W 表示可读可写，
－0 表示复位后的初始化值为 0，－1 表示复位后的初始化值为 1)

图 2-6　BSCR 寄存器

表 2-3　BSCR 寄存器各位代表的含义

BIT		RESET VALUE (复位值)	FUNCTION （功能）
NO.	NAME		
15-12	BNKCMP	111	Bank Compare. Detemines the extemal memory-bank size. BNKCMP is used to mask the four MSBs of an address. For example,if Bnkcmp=1111b, the four MSBs (bits 12-15) are compared, resulting in a bank size of 4K words. Bank sizes of 4K words are allowed 块屏蔽位域（Bank Compare）。BNKCMP 通过屏蔽十六位地址的高四位来决定外部存储区块的大小。例如，当 BNKCMP=1111 时，外部存储区块大小为 0x0000 至 0x0FFF。外部存储区块的大小最大为 64 KW
11	PS-DS	1	Program read-data read access. Inserts an extra cycle between conseacutive accesses of program read and data read or data read and program read. PS-DS=0　No extra cycles are inseried by this feature. PS-DS=1　One extra cycle is inserted between consecutive data and program reads 程序区和数据区的读访问。本位指示在两个连续的程序空间读操作或是在数据空间读操作，或是在一个数据读操作和在一个程序读操作期间插入一个额外的时钟周期。当 PS-DS=0 时，不插入这个时钟周期；当 PS-DS=1 时，插入一个时钟周期
10-3	Reserved	0	These bits are reserved and are unaffected by writes 这个位域保留，不受写操作影响

续表

BIT		RESET VALUE (复位值)	FUNCTION (功能)
NO.	NAME		
2	HBH	0	HPI Bus holder. Controls the HPI bus holder feature. HBH is cleared to 0 at reset. HBH=0　The bus holder is disabled. HBH=1　The bus holder is enabled. When not driven, the HPI data bus (HD[7:0]) is held in the previous logic hevel HPI 总线保持。这个位控制 HPI 总线的保持特性。芯片复位时 HPH 置 0。当 HPH=0 时，不具有总线保持特性；当 HPH=1 时，启动总线保持特性，HPI 总线上没有出现新的驱动时序时，总线上将保持前一个时序的逻辑电平
1	BH	0	Bus holder. Controls the data bus holder feature. BH is cleared to 0 at reset. BH=0　The bus holder is disabled. BH=1　The bus holder is enabled. When not driven, the data bus (D[15:0]) is held in the previous logic level 数据总线保持。这个位控制数据总线的保持特性。芯片复位时 BH 置 0。当 BH=0 时，不具有总线保持特性；当 BH=1 时，启动总线保持特性，数据总线上没有出现新的驱动时序时，总线上将保持前一个时序的逻辑电平
0	EXIO	0	Extemal bus interface off. The EXIO bit controls the extemal bus-off function. EXIO=0　The extemal bus interface functions as usual. EXIO=1　The address bus, data bus, and control signals become inactive after complelting the current bus cycle. Note that the DROM, MP/MC, and OVLY bits in the PMST and the HM bit of ST1 cannot be modified when the interface is disabled 外部总线接口开关控制位。当 EXIO=0 时，外部总线接口工作正常；当 EXIO=1 时，当前总线周期完成后，地址总线、数据总线和控制信号进入不活跃状态，在这种状态下，PMST 寄存器中的 DROM、MP/MC 和 OVLY 位以及 ST1 寄存器中的 HM 位不能被修改

3．并行 I/O 口

VC5402 具有 64 K×16 bit 的外部 I/O 访问空间，并且提供了专用于 I/O 空间访问的 C/C++语句，详见第三章“C/C++程序设计”。所有 C5000 系列的 64 K×16 bit 的 I/O 空间都是指访问控制范围的空间，只有在通过并行 I/O 口与外部资源相连接时才有实在的意义，否则，这个空间是不能用来存取的。也就是说，对这个空间的访问是这样进行的：当读这个空间时，实际上是从与外部并口相连接的器件输出缓冲区读出数据；当写数据到这个空间时，实际上是写入到与外部并口相连接的器件输入缓冲区中。I/O 空间如图 2-7 所示。

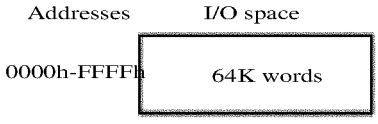


图 2-7　I/O 空间

4．直接内存访问（DMA）控制器

DMA 是指在不需要 CPU 干预的情况下在映射存储区的不同区域间传输数据的技术。DMA 支持片上存储器、片上外设（包括 EHPI8）或是外部设备间无 CPU 负责的独立数据交换。它具有以下特性：

第一，后台操作，即 DMA 操作和 CPU 处理是相互独立的，DMA 由 DMA 控制器管理。

第二，六个通道，即 DMA 具有六个独立的传输通道，可以用于传送完全不同的数据。

第三，HPI 接口访问，即 HPI8 借助 DMA 总线在不需要 CPU 干预的情况下与映射存储区进行数据交换，这使得 HPI8 可以访问整个的映射存储区，成为增强的主机处理器接口（EHPI8）。

第四，多帧数据交换，即 DMA 可以完成数据块的直接交换处理。

第五，可编程的优先级，每个通道可以通过编程设定其优先级。

第六，可编程的地址发生模式，即每个通道的读写传输的源或目标地址寄存器可以通过编程的方法设置其在被访问之后，或是自动地址加一，或是地址保持不变。当传输连续的数据块时，采用源或目标地址自动加一的方法十分有效；在传输独立的数据或传输块数据结束后，采用源或目标地址保持不变的方法。

第七，全地址访问权限，即 DMA 可以访问片上存储器、片上外设和外部存储器（视器件不同而不同）。

第八，DMA 可以通过编程设置传输数据的格式，每个通道传送完毕后具有自动初始化的能力。传输数据可以和选择的事件同步，数据帧或数据块传输完毕后，每个 DMA 通道可以向 CPU 发送一个中断信号。

对 DMA 的配置和操作是通过设置相关的内存映射控制寄存器来完成的。这里应强调指出，内存映射控制寄存器并不是 DMA 真正的控制寄存器。这对于 C5000 系列的其它映射寄存器是一样的道理，映射的寄存器只是一个桥梁，即通过写控制字到映射的寄存器去，器件会自动将这些控制字写入到相应的物理控制寄存器中去，即映射的寄存器只提供了一种访问内部物理控制寄存器的方法。所以，可以有很多物理寄存器同时映射到同一个映射寄存器位置处，再通过一个子地址寄存器来区分具体的物理寄存器。使用这种方法，用两个存储器映射寄存器就可以确定很多的物理寄存器位置，DMA 就是采用了这种映射方法。

图 2-8 中左边的 DMSDI、DMSDN 和 DMSA 为右边很多物理寄存器的存储器映射寄存器。可见，图中通过对三个映射寄存器的访问就可达到对很多 DMA 物理寄存器访问的目的。具体的访问过程是这样的：要访问 DMA 的某一物理寄存器，首先要写一个地址数据到子段地址寄存器（DMSA），这个地址数据对于 DMA 来说是有固定范围的（见表 2-4），这个操作表明设置了一个指向实际要访问的 DMA 物理寄存器地址的指针；然后，写数据值到子

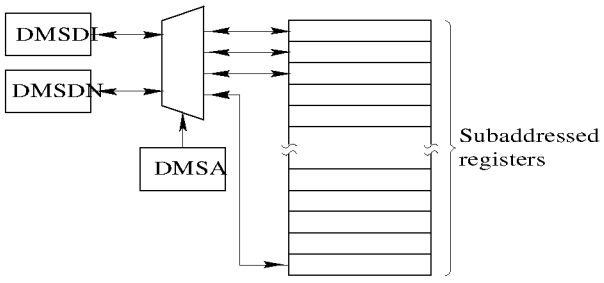


图 2-8　DMA 物理寄存器与映射寄存器

段地址访问寄存器（DMSDI 或 DMSDN）中，相当于向指针指向的要访问的 DMA 物理寄存器写入控制数据。为了操作上的灵活性，DMA 提供了两种类型的子段地址访问寄存器：一种是子地址被访问后自动加一的 DMSDI 寄存器，另一种是访问前后子地址保持不变的 DMSDN 寄存器。

表 2-4　DMA 映射控制寄存器*

Address (地址)	SubAddress (子地址)	Name (名称)	Function (功能)
54h	—	DMPREC	Channel Priority and Enable Control Register 通道优先级和使能控制寄存器
55h	—	DMSA	Subbank Address Register 子段地址寄存器
56h	—	DMSDI	Subbank Access Register With Autoincrement 地址自增模式下的子段访问寄存器
57h	—	DMSDN	Subbank Access Register Without Autoincrement 非地址自增模式下的子段访问寄存器
—	00h	DMSRC0	Channel 0 Source Address Register 0 通道源地址寄存器
—	01h	DMDST0	Channel 0 Destination Address Register 0 通道目的地址寄存器
—	02h	DMCTR0	Channel 0 Element Count Register 0 通道单元计数器寄存器
—	03h	DMSFC0	Channel 0 Sync Select and Frame Count Register 0 通道同步选择和帧计数器寄存器
—	04h	DMMCR0	Channle 0　Transfer Mode Control　Register 0 通道传输模式控制寄存器
—	05h	DMSRC1	Channel 1 Source Address Register 1 通道源地址寄存器
—	06h	DMDST1	Channel 1 Destination Address Register 1 通道目的地址寄存器
—	07h	DMCTR1	Channel 1 Element Count Register 1 通道单元计数器寄存器
—	08h	DMSFC1	Channel 1 Sync Select and Frame Count Register 1 通道同步选择和帧计数器寄存器
—	09h	DMMCR1	Channel 1 Transfe Mode Control Register 1 通道传输模式控制寄存器
—	0Ah	DMSRC2	Channel 2 Source Address Register 2 通道源地址寄存器

续表

Address (地址)	SubAddress (子地址)	Name (名称)	Function (功能)
—	0Bh	DMDST2	Channel 2 Destination Address Register 2 通道目的地址寄存器
—	0Ch	DMCTR2	Channel 2 Element Count Register 2 通道单元计数器寄存器
—	0Dh	DMMCR2	Channel 2 Sync Select and Frame Count Register 2 通道同步选择和帧计数器寄存器
—	0Eh	DMMCR2	Channel 2 Transfe Mode Control Register 2 通道传输模式控制寄存器
—	0Fh	DMSRC3	Channel 3 Source Address Register 3 通道源地址寄存器
—	10h	DMDST3	Channel 3 Destination Address Register 3 通道目的地址寄存器
—	11h	DMCTR3	Channel 3 Element Count Register 3 通道单元计数器寄存器
—	12h	DMSFC3	Channel 3 Sync Select and Frame Count Register 3 通道同步选择和帧计数器寄存器
—	13h	DMMCR3	Channel 3 Transfe Mode Control Register 3 通道传输模式控制寄存器
—	14h	DMSRC4	Channel 4 Source Address Register 4 通道源地址寄存器
—	15h	DMDST4	Channel 4 Destination Address Register 4 通道目的地址寄存器
—	16h	DMCTR4	Channel 4 Element Count Register 4 通道单元计数器寄存器
—	17h	DMSFC4	Channel 4 Sync Select and Frame Count Register 4 通道同步选择和帧计数器寄存器
—	18h	DMMCR4	Channel 4 Transfe Mode Control Register 4 通道传输模式控制寄存器
—	19h	DMSRC5	Channel 5 Source Address Register 5 通道源地址寄存器
—	1Ah	DMDST5	Channel 5 Destination Address Register 5 通道目的地址寄存器
—	1Bh	DMCTR5	Channel 5 Element Count Register 5 通道单元计数器寄存器

续表

Address (地址)	SubAddress (子地址)	Name (名称)	Function (功能)
—	1Ch	DMSFC5	Channel 5 Sync Select and Frame Count Register 5 通道同步选择和帧计数器寄存器
—	1Dh	DMMCR5	Channel 5 Transfe Mode Control Register 5 通道传输模式控制寄存器
—	1Eh	DMSRCP	Source Program Page Address (all channels) 源程序页地址（适用于所有通道）
—	1Fh	DMDSTP	Destination Program Page Address (all channels) 目的程序页地址（适用于所有通道）
—	20h	DMIDX0	Element Address Index Register 0 单元地址索引寄存器 0
—	21h	DMIDX1	Element Address Index Register 1 单元地址索引寄存器 1
—	22h	DXFR10	Frame Address Index Register 0 帧地址索引寄存器 0
—	23h	DMFR11	Frame Address Index Register 1 帧地址索引寄存器 1
—	24h	DMGSA	Global Source Address Reload Register 总体源地址再装入寄存器
—	25h	DMGDA	Global Destination Address Reload Register 总体目的地址再装入寄存器
—	26h	DMGCR	Global Element Count Reload Register 总体单元计数器再装入寄存器
—	27h	DMGFR	Global Frame Count Reload Register 总体帧计数器再装入寄存器

*表 2-4 来自 TI 的技术资料，引用于此仅供参考。为方便读者，作者加了中译文，供参考。

结合表 2-4 可用 C 语句说明具体访问的方法。这种 C 语句的详细说明请参考第三章“C/C++程序设计”部分。

```
#define   DMSA        0x55
#define   DMSDN       0x57
#define   DMSRC5      0x19
```

以上三个 C 定义语句应定义为全局地址变量，即应放在主程序前面，或放在头文件中。上面三句中变量的含义请参考表 2-4。

```
*(short *)DMSA   =   DMSRC5;
*(short *)DMSDN   =   0x1000;
```

上面这两个语句选中了 DMA 通道 5 的源地址寄存器，并初始化为数值 0x1000，DMA 寄存器地址保持不变。使用同样方法可以访问 DMSDI 寄存器，并使 DMA 寄存器的地址在被访问后自动加一。可以用这种方法按顺序从第一个 DMA 控制寄存器开始，不需要设置 DMA 寄存器地址，直接写入初始化控制数据，即可将 DMA 寄存器全部初始化完毕。

DMA 六个通道的优先级、工作及中断由通道优先工作控制寄存器（DMPREC）来管理。DMPREC 位于数据空间的 0054h 地址处，上电初始化时的值为 0000h。DMPREC 寄存器及其各位的含义如图 2-9 和表 2-5 所示。DMA 在数据传输完毕后会向 CPU 发送中断信号，DMA 中断信号与其它片上外设的中断信号复用相同的中断向量位置，可以节约中断资源占用的寄存器存储区域。DMPREC 可用于选择中断。对于 VC5402，其 DMA 通道中断向量表如表 2-6 所示。

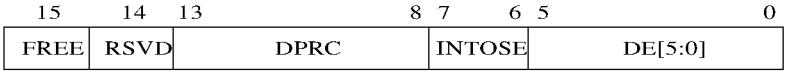


图 2-9　DMPREC 寄存器

表 2-5　DMPREC 寄存器各位的含义*

BIT		Reset Value (复位值)	Function （功能）
NO.	Name		
15	FREE	0	This bit controls the behavior of the DMA controller during emulation. When FREE=0, DMA transfers are suspended when the emulator strops. When FREE=1, DMA transfers continue even during emulation stop 该位控制了仿真过程中 DMA 控制器的行为。当 FREE=0 时，当仿真停止时 DMA 传输被挂起（即暂停）；当 FREE=1 时，即使仿真过程停止，DMA 传输继续进行
14	RSVD	0	Reserved. Values written to this field have no effect 保留。本位不受写操作的影响
13	DPRC[5]	0	DMA channel 5 priority control bit. DPRC[5]=1　High priority DPRC[5]=0　Low priority DMA 通道 5 优先级控制位。当 DPRC[5]=1 时，为高优先级；当 DPRC[5]=0 时，为低优先级
12	DPRC[4]	0	DMA channel 4 priority control bit. DPRC[4]=1　High priority DPRC[4]=0　Low priority DMA 通道 4 优先级控制位。当 DPRC[4]=1 时，为高优先级；当 DPRC[4]=0 时，为低优先级
11	DPRC[3]	0	DMA channel 3 priority control bit. DPRC[3]=1　High priority DPRC[3]=0　Low priority DMA 通道 3 优先级控制位。当 DPRC[3]=1 时，为高优先级；当 DPRC[3]=0 时，为低优先级

续表

BIT		Reset Value (复位值)	Function (功能)
NO.	Name		
10	DPRC[2]	0	DMA channel 2 priority control bit. DPRC[2]=1　High priority DPRC[2]=0　Low priority DMA 通道 2 优先级控制位。当 DPRC[2]=1 时，为高优先级；当 DPRC[2]=0 时，为低优先级
9	DPRC[1]	0	DMA channel 1 priority control bit. DPRC[1]=1　High priority DPRC[1]=0　Low priority DMA 通道 1 优先级控制位。当 DPRC[1]=1 时，为高优先级；当 DPRC[1]=0 时，为低优先级
8	DPRC[0]	0	DMA channel 0 priority control bit. DPRC[0]=1　High priority DPRC[0]=0　Low priority DMA 通道 0 优先级控制位。当 DPRC[0]=1 时，为高优先级；当 DPRC[0]=0 时，为低优先级
7-6	INTOSEL	0	Interrupt muitiplex control bits. The INTOSEL bits control how the DMA interrupts will be assigned in the interrupt vector table and IMR/IMF registers. The effects of this field on the operation are device-specific (refer to Table 3-3,Table 3-4, and Table 3-5.) 中断选择控制位域。INTOSEL 控制了 DMA 中断如何被分配到中断向量表和 IMR/IFR 寄存器中，对于不同的芯片，这个位域的作用不同
5	DE[5]	0	DMA channel 5 enable bit. DE[5]=1　Enables DMA channel 5 DE[5]=0　Disables DMA channel 5 DMA 通道 5 使能位。当 DE[5]=1 时，开启 DMA 通道 5；当 DE[5]=0 时，关闭 DMA 通道 5
4	DE[4]	0	DMA channel 4 enable bit. DE[4]=1　Enables DMA channel 4 DE[4]=0　Disables DMA channel 4 DMA 通道 4 使能位。当 DE[4]=1 时，开启 DMA 通道 4；当 DE[4]=0 时，关闭 DMA 通道 4
3	DE[3]	0	DMA channel 3 enable bit. DE[3]=1　Enables DMA channel 3 DE[3]=0　Disables DMA channel 3 DMA 通道 3 使能位。当 DE[3]=1 时，开启 DMA 通道 3；当 DE[3]=0 时，关闭 DMA 通道 3

续表

BIT		Reset Value (复位值)	Function (功能)
NO.	Name		
2	DE[2]	0	DMA channel 2 enable bit. DE[2]=1　Enables DMA channel 2 DE[2]=0　Disables DMA channel 2 DMA 通道 2 使能位。当 DE[2]=1 时，开启 DMA 通道 2；当 DE[2]=0 时，关闭 DMA 通道 2
1	DE[1]	0	DMA channel 1 enable bit. DE[1]=1　Enables DMA channel 1 DE[1]=0　Disables DMA channel 1 DMA 通道 1 使能位。当 DE[1]=1 时，开启 DMA 通道 1；当 DE[1]=0 时，关闭 DMA 通道 1
0	DE[0]	0	DMA channel 0 enable bit. DE[0]=1　Enables DMA channel 0 DE[0]=0　Disables DMA channel 0 DMA 通道 0 使能位。当 DE[0]=1 时，开启 DMA 通道 0；当 DE[0]=0 时，关闭 DMA 通道 0

*表 2-5 是来自 TI 的技术资料，本文原文引用。为方便读者，作者加了中译文，供参考。

表 2-6　VC5402 的 DMA 通道中断向量表*

中断号	INTOSEL[1：0]值			
（IMR/IFR#）	00b	01b	10b	11b
7	计时器 1 中断	计时器 1 中断	DMA 通道 1 中断	保留
10	McBSP1 接收数据中断	DMA 通道 2 中断	DMA 通道 2 中断	保留
11	McBSP1 发送数据中断	DMA 通道 3 中断	DMA 通道 3 中断	保留

*表 2-6 来自 TI 技术资料，原文引用。

VC5402 的 DMA 中断类型是由通道模式控制寄存器（DMMCR）中的 IMOD 和 DINM 位来决定的，有效的中断模式列于表 2-7 中。

表 2-7　DMA 向 CPU 发出的中断模式*

模式	DINM	IMOD	发出中断情况
ABU（非自增）	1	0	仅当缓冲区满时
ABU（非自增）	1	1	当缓冲区半满和满时
多重帧	1	0	当块传输完成时（要求 DMCTRn=DMSEFCn[7:0]=0）
多重帧	1	1	在数据帧或数据块传输完成时（DMCTRn=0）
其它	0	X	无中断产生
其它	0	X	无中断产生

*表 2-7 来自 TI 技术资料，原文引用。

限于篇幅，表 2-4 中的其他控制寄存器就不作详细介绍了。理解这些控制寄存器有助于理解 DMA 的工作机理。读者也可参阅 TI 的相关技术文档，进一步了解有关内容。

下面进一步讨论 VC5402 的 DMA 访问存储器映射。如表 2-8 所示，VC5402 仅能访问映射数据存储器的一部分，且不受 MP/MC、DROM 和 OVLY 等控制位的影响。每个 16 bit 的 DMA 数据的传输需要一先读后写相继的操作；VC5402 仅支持片内 DATA 区的数据传输，需要 4 个时钟周期。

表 2-8　VC5402 的 DMA 映射存储器*

Address Range(地址范围) (Hex) (十六进制)		Description(描述)
Program space (程序区)	00 0000 – 0F FFFF	Reserved(保留)
Data space (数据区)	0000 – 001F	Reserved(保留)
	0020	McBSP0 data receive register (DRR20) McBSP0 数据接收寄存器（DRR20）
	0021	McBSP0 data receive register (DRR10) McBSP0 数据接收寄存器（DRR10）
	0022	McBSP0 data transmit register (DXR20) McBSP0 数据发送寄存器（DXR20）
	0023	McBSP0 data transmit register (DXR10) McBSP0 数据发送寄存器（DXR10）
	0024 – 003F	Reserved(保留)
	0040	McBSP1 data receive register (DRR21) McBSP1 数据接收寄存器（DRR21）
	0041	McBSP1 data receive register (DRR11) McBSP1 数据接收寄存器（DRR11）
	0042	McBSP1 data transmit register (DXR21) McBSP1 数据发送寄存器（DXR21）
	0043	McBSP1 data transmit register (DXR11) McBSP1 数据发送寄存器（DXR11）
	0044 – 005F	Reserved(保留)
	0060 – 007F	Scratchpad RAM(临时 RAM)
	0080 – 3FFF	Daram(双访问 RAM)
	4000 – FFFF	Reserved(保留)
I/O space(I/O 区)	0000 – FFFF	Reserved(保留)

*表 2-8 原文引用自 TI 的技术资料，其中作者加了中文注释供参考。

DMMCR 寄存器中的 CTMOD 位域的值控制了 DMA 每个通道的传输计数模式。在多帧模式下，每次传输完毕后，可以根据传输的元素或帧的序号修改源地址和目标地址，这对于连续地传输数据块是很方便的。另一种模式是自动缓存模式，即 ABU 模式。在这种模式下，每个通道元素的计数值代表了缓冲区的大小，在传输数据过程中，这个值也不会被

改变。下面以 ABU 模式下，VC5402 的 DMA 将多通道缓冲串口 0（McBSP0）接收到的数据搬移到数据区为例，介绍对 DMA 的具体操作方法，这种方法的数据搬移特别适合于后续的 FFT 等数据处理算法。

下面为用 C 语句给出 DMA 寄存器的全局定义。为了阅读方便，变量名与表 2-4 中的相一致。

```
#define   DMPREC    0x54
//Channel Priority and Enable Control Register 定义了 DMPREC 的存储器映射地址
#define   DMSA      0x55
//Sub-bank Address Register   定义了 DMSA 的存储器映射地址，以下均是指定地址的定义
#define   DMSDI     0x56
//Sub-bank data Regitster with autoincrement
#define   DMSDN     0x57
//Sub-bank Data Register without modification
#define   DMSRC0    0x00
//Channel 0 Source Address Register
#define   DMDST0    0x01
//Channel 0 Destination Address Register
#define   DMCTR0    0x02
//Channel 0 Element Count Register
#define   DMSFC0    0x03
//Channel 0 Sync Select and Frame Count Register
#define   DMMCR0    0x04
//Channel 0 Transfer Mode Control Register
#define   DMSRC1    0x05
//Channel 1 Source Address Register
#define   DMDST1    0x06
//Channel 1 Destination Address Register
#define   DMCTR1    0x07
//Channel 1 Element Count Register
#define   DMSFC1    0x08
//Channel 1 Sync Select and Frame Count Register
#define   DMMCR1    0x09
//Channel 1 Transfer Mode Control Register
#define   DMSRC2    0x0A
//Channel 2 Source Address Register
#define   DMDST2    0x0B
//Channel 2 Destination Address Register
#define   DMCTR2    0x0C
//Channel 2 Element Count Register
```

```
#define   DMSFC2     0x0D
//Channel 2 Sync Select and Frame Count Register
#define   DMMCR2     0x0E
//Channel 2 Transfer Mode Control Register
#define   DMSRC3     0x0F
//Channel 3 Source Address Register
#define   DMDST3     0x10
//Channel 3 Destination Address Register
#define   DMCTR3     0x11
//Channel 3 Element Count Register
#define   DMSFC3     0x12
//Channel 3 Sync Select and Frame Count Register
#define   DMMCR3     0x13
//Channel 3 Transfer Mode Control Register
#define   DMSRC4     0x14
//Channel 4 Source Address Register
#define   DMDST4     0x15
//Channel 4 Destination Address Register
#define   DMCTR4     0x16
//Channel 4 Element Count Register
#define   DMSFC4     0x17
//Channel 4 Sync Select and Frame Count Register
#define   DMMCR4     0x18
//Channel 4 Transfer Mode Control Register
#define   DMSRC5     0x19
//Channel 5 Source Address Register
#define   DMDST5     0x1A
//Channel 5 Destination Address Register
#define   DMCTR5     0x1B
//Channel 5 Element Count Register
#define   DMSFC5     0x1C
//Channel 5 Sync Select and Frame Count Register
#define   DMMCR5     0x1D
//Channel 5 Transfer Mode Control Register
#define   DMSRCP     0x1E
//Source Program Page Address
#define   DMSTP      0x1F
//Destination Program Page Address
#define   DMIDX0     0x20
```

```
//Element Address Index Register 0
#define   DMIDX1      0x21
//Element Address Index Register 1
#define   DMFRI0      0x22
//Frame Address Index Register 0
#define   DMFRI1      0x23
//Frame Address Index Register 1
#define   DMGSA       0x24
//Global Source Address Reload Register
#define   DMGDA       0x25
//Global Destination Address Reload Register
#define   DMGCR       0x26
//Global Element Count Reload Register
#define   DMGFR       0x27
//Global Frame Count Reload Register
```

上面的程序段对 DMA 所有的寄存器都作了访问定义，在程序中不一定都会用到。下面的程序段给出了 VC5402 通过 DMA 控制器从 McBSP0 数据接收端传输 16 bit 的数据字到数据映射存储区的 DMA 的配置；传输模式为 ABU 模式；源地址为 McBSP0 数据接收寄存器（DRR10），目标缓冲区为数据空间的 03000h-030FFh 容量为 100h 的按字计算的区域。同步事件为 McBSP0 接收事件，使用了 DMA 通道#1。

```
*(short   *)DMSA=DMSRC1;     //set source address to DRR10
*(short   *)DMSDN=DRR1_0;
*(short   *)DMSA=DMDST1;     //set destination address to 3000
*(short   *)DMSDN=0x3000;
*(short   *)DMSA=DMCTR1;     //set buffer size to 100h words
*(short   *)DMSDN=0x100;
*(short   *)DMSA=DMSFC1;
*(short   *)DMSDN=0x8000;
/*0001~~~~~~~~~~~~ (DSYN)   McBSP0 receive sync event
 ~~~~0~~~~~~~~~~~ (DBLW)   Single-word mode
 ~~~~~000~~~~~~~~          Reserved
 ~~~~~~~~00000000 (Frame Count) Frame count   is not relevant in ABU mode*/
*(short   *)DMSA=DMMCR1;
*(short   *)DMSDN=0x504C;
/* 0~~~~~~~~~~~~~ (AUTOINIT) Auto-initialization disabled
   ~1~~~~~~~~~~~~ (DINM)     DMA Interrupts enabled
 ~~0~~~~~~~~~~~~ (IMOD)     Interrupt at full buffer
~~~1~~~~~~~~~~~~ (CTMOD)  ABU (non-decrement) mode
```

```
~~~~0~~~~~~~~~~~~          Reserved
~~~~~000~~~~~~~~ (SIND)    No modify on source address (DRR10)
~~~~~~~~01~~~~~~ (DMS)     Source in data space
~~~~~~~~~~0~~~~~           Reserved
~~~~~~~~~~~011~~ (DIND)    Post increment destination address with DMIDX0
~~~~~~~~~~~~~~01 (DMD)     Destination in data space
*(short   *)DMSA=DMIDX0;   //set element address index to +1
*(short   *)DMSDN=0x0001;
*(short   *)DMPREC=0x0202;
/* 0~~~~~~~~~~~~~~~ (FREE) DMA stops on emulation stop
~0~~~~~~~~~~~~~~           Reserved
~~0~~~~~~~~~~~~~ (DPRC[5]) Channel 5 low priority
~~~0~~~~~~~~~~~~ (DPRC[4]) Channel 4 low priority
~~~~0~~~~~~~~~~~ (DPRC[3]) Channel 3 low priority
~~~~~0~~~~~~~~~~ (DPRC[2]) Channel 2 low priority
~~~~~~1~~~~~~~~~ (DPRC[1]) Channel 1 high priority
~~~~~~~0~~~~~~~~ (DPRC[0]) Channel 0 low priority
~~~~~~~~00~~~~~~ (INTOSEL)N/A
~~~~~~~~~~0~~~~~ (DE[5]) Channel 5 disabled
~~~~~~~~~~~0~~~~ (DE[4]) Channel 4 disabled
~~~~~~~~~~~~0~~~ (DE[3]) Channel 3 disabled
~~~~~~~~~~~~~0~~ (DE[2]) Channel 2 disabled
~~~~~~~~~~~~~~1~ (DE[1]) Channel 1 disabled
~~~~~~~~~~~~~~~0 (DE[0]) Channel 0 disabled   */
```

该程序在 Simulator 下仿真查看不到寄存器的值，必须借助 Emulator 才行。

5．增强的 8 bit 主机接口（HPI8）

增强的 8 bit 主机接口（EHPI8）将 DSP 设置为从模式来完成与主处理机之间的通信任务，使得主机和 DSP 均可以访问 DSP 片上存储器。这个接口包括 8 条双向数据线和多条控制线，如图 2-10 所示。8 条数据线（HD0-HD7）用于与主机交换数据，两个控制输入端（HCNTL0 和 HCNTL1）用于主机指定访问 HPI8 的控制寄存器 HPIC、HPI8 数据寄存器（HPID）或 HPI8 地址寄存器（HPIA）；HBIL 管脚用于指定 16 bit 数据传输时的高 8 bit 和低 8 bit；HPID 寄存器具有自动地址加一的访问特性，即可以从连续的数据空间读出或写入数据块，在这个连续的读或写过程中，HPIA 寄存器是自动增一的。

EHPI8 通过中断实现主机和 DSP 之间通信的握手协议。主机可以通过写 HPIC 的 DSPINT 位为 1 去中断 DSP 的空闲状态，DSP 通过 HINT 管脚向主机发送中断请求。DSP 通过写 HPIC 的 HINT 位为 1 将 HINT 管脚低中断信号送给主机，主机确认后再次写 HPIC 寄存器 HINT 位将 HINT 置 1，从而完成主机接收 DSP 中断信号的握手。具有单一或多数据探针、带有或不带有地址锁存启动（或“使能”）信号的主机设备都可以与 DSP 的 EHPI8

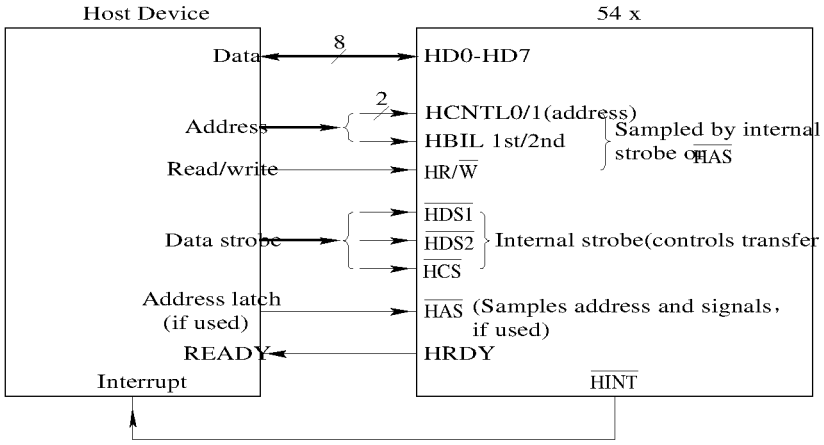


图 2-10 HPI8 与主机的通信

相连接。HPI8 的寄存器及 HPIC 寄存器的各位含义如表 2-9 和表 2-10 所示。主机将 HPIA 寄存器视为一个地址指针，借助于 HPIA 主机可以访问 VC5402 的全部片上存储器。图 2-11 为 VC5402 的 HPI 访问映射存储器，实际上这个存储器只针对物理双访问 DRAM 区才有意义。

表 2-9 HPI8 寄存器

Namd (名称)	'54x Address (54×系列地址)	Description (描述)
HPIA	—	HPI address register. Directly accessible only by the host. Contains the address in the '54x on-chip RAM where the current access occurs HPI 地址寄存器。这个寄存器只能被主机直接访问。当主机访问时，这个寄存器存放被访问的地址
HPIC	002Ch	HPI control register. Directly accessible by either the host or by the '54x. Contains control and status bits for HPI-8 operations HPI 控制寄存器。这个寄存器既可以被主机直接访问，又可以被'54x 系列 DSP 片上 CPU 访问。对 HPI-8 接口来说，这个寄存器包括控制位和状态位
HPID	—	HPI data register. Ddirectly accessible only by the host. Contains data that is read from the '54x on-chip memory if the current access is a read, or data that is written to on-chip memory if the current access is a write HPI 数据寄存器。这个寄存器仅能被主机直接访问。当主机访问时，这个寄存器存放被访问的数据。与 HPIA 联合使用

表 2-10　HPIC 寄存器各位的含义

Bit(位)	Reset Value (复位值)	Function (功能)
BOB	0	Byte-order bit. This bit determines the placement for the two bytes of a transfer. If BOB=1, the first byte of a transfer is least significant. If BOB=0, the first byte is most significant. This bit can only be accessed (written or read) by the host, and if must be initialized before the first data or address register access 字节传送顺序控制位。在双字节数据传输中，这一位的值决定传送数据或地址的字节排列方式。当 BOB=0 时，先传送最低位；当 BOB=1 时，先传送最高位。这一位仅能被主机访问，并且进行数据或地址的传输前必须要先对它进行初始化
DSPINT	0	Host-to-'54x interrupt. When the host writes a 1 to this bit, a '54x interrupt is generated. The bit can only be written to by the host, and is always read as 0 by both the host and the '54x. When the host writes to HPIC, both bytes must write the same value. For a detailed description of the DSPINT function, see section 4.5 on page 4-23 主机诱导 DSP 中断控制位。这一位只能由主机写入，当主机将这一位置 1 时，一个 DSP 中断将会产生。这一位可以被主机和 DSP 读出，但读出的值始终为 0
HINT	0	'54x-to-host interrupt. This bit determines the state of the '54x HINT output, which can be used to interrupt the host. When the HINT bit is set to 1, the HINT output is driven low, and when the bit is cleared to 0, the output is driven high. The host clears the bit by writing a 1 to it. For a detailed description of the HINT function, see section 4.5 on page 4-23 DSP 驱动主机中断控制位。这一位的值决定了 DSP 的 HINT 管脚输出的电平状态，被用来中断主机。当 HINT 位置 1 时，HINT 管脚输出低电平；当 HINT 位置 0 时，HINT 管脚输出高电平。HINT 位仅能由 DSP 置位，也仅能由主机向这一位写 1 清空
XHPIA	X	Extended address enable. When XHPIA=1, host writes to the HPIA register are loaded into the most significant bits HPIA[n:16]. If XHPIA=0, host writes to HPIA are loaded into HPIA[15:0]. All n+1 address bits are incremented in the autoincrement mode. Reading the HPIA register is performed in the same manner. Only the host has access to this bit 扩展地址使能位。当 XHPIA=1 时，主机将访问 HPIA 的高位域 HPIA[n:16]〈n>16〉；当 XHPIA=0 时，主机将访问 HPIA 的低位域 HPIA[15:0]
HPIENA	X	HPI enable status bit. This bit latches the reset value of the HPIENA pin, and can be used by the '54x to determine if the HPI-8 is enabled or disabled. This bit is not affected by writes, and is not available to the host. Note: This bit is not available on all devices. For more details, see the specific HPIC register diagrams that follow HPIA 使用状态位。这一位锁定在复位时 HPIENA 的电平上，不受写操作的影响，而且主机不能访问。对于 C54 系列 DSP 来说，这一位用于决定 HPI-8 工作模式是否可用

注：为方便读者阅读，表中特别给出了中文参考。若直接给出中文，怕不够准确，影响读者理解。

HPI8 读写访问包括两部分：其一是主机与 HPI8 寄存器之间的数据交换，为外部数据交换；其二是内部数据交换，即 HPI8 与片上 RAM 之间的数据交换。通常内部数据交换借助 DMA 总线来完成，受 DMA 控制器控制，不需要 CPU 干预，且内部数据交换为 16 bit 的数据字交换。外部数据交换为 8 bit 的字节交换，在这个交换过程中，主机控制 HBIL 为低电平时传送第一个字节，然后控制 HBIL 为高电平传送第二个字节。如果在这个过程中，主机发生了中断，则该数据会丢失，为避免出现这种情况，在传送数据的过程中主机应不间断地访问 HBIL 以确保其极性正确。对于每个字节的传送，主机使用 HPI 口的数据探针脚（HDS1 和 HDS2）及数据选择脚（HCS）来控制对 VC5402 的访问时期。

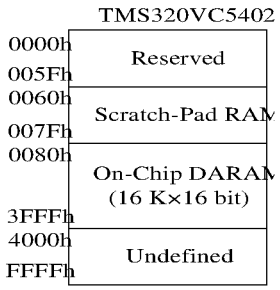


图 2-11　VC5402 的 HPI 访问存储区

6．多通道缓冲串行口（McBSPs）

需要首先指出的是 McBSP 和 HPI8 都可以设置为通用 I/O 口作为 DSP 的控制信号，具体的设置方法在此不作详细说明。VC5402 具有两个 McBSPs，每个 McBSP 具有表 2-11 所示的接口信号，其内部结构如图 2-12 所示。

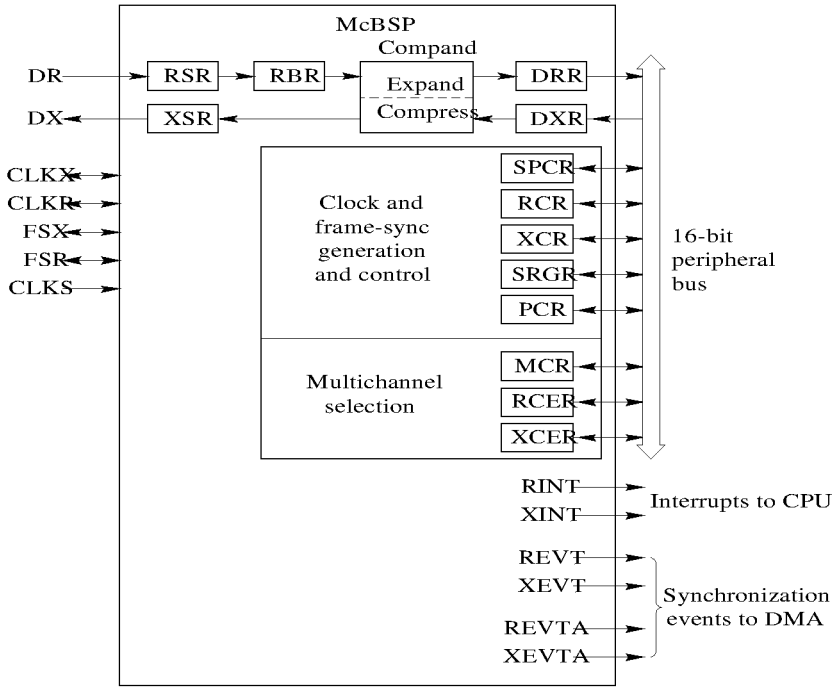


图 2-12　McBSP 内部结构

表 2-11　McBSP 接口信号

Pin (管脚名)	I/O/Z (输入/输出/高阻)	Description (含义)
CLKR	I/O/Z	Receive clock(接收时钟)
CLKX	I/O/Z	Transmit clock(发送时钟)
CLKS	I	External clock(使用外部时钟)
DR	I	Received serial data(接收串行数据)
DX	O/Z	Transmitted serial data(发送串行数据)
FSR	I/O/Z	Receive frame synchronization(接收帧同步信号)
FSX	I/O/Z	Transmit frame synchronization(发送帧同步信号)

注：I=Input, O=Output, Z=High-impedance

访问 McBSP 的方法与访问 DMA 的方法相类似，McBSP 寄存器映射的方法也是采用了子地址区分的多址映射一址的方法。表 2-12 给出了 McBSP 映射寄存器的地址及含义，并结合表 2-12 给出了访问 McBSP 寄存器的方法。限于篇幅，本文没有详细介绍各个寄存器中每位的含义。

表 2-12　McBSP 映射寄存器

十六进制地址			子地址 （十六进制）	寄存器名	含　义
McBSP0	McBSP1	McBSP2			
-	-	-		RBR[1,2]	McBSP 接收缓冲寄存器 1 和 2
-	-	-		RSR[1,2]	McBSP 接收移位寄存器 1 和 2
-	-	-		XSR[1,2]	McBSP 发送移位寄存器 1 和 2
0020	0040	0030	-	DRR2x	McBSP 数据接收寄存器 2
0021	0041	0031	-	DRR1x	McBSP 数据接收寄存器 1
0022	0042	0032	-	DXR2x	McBSP 数据发送寄存器 2
0023	0043	0033	-	DXR1x	McBSP 数据发送寄存器 1
0038	0048	0034	-	SPSAx	McBSP 子地址寄存器
0039	0049	0035	0000	SPCR1x	McBSP 串口控制寄存器 1
0039	0049	0035	0001	SPCR2x	McBSP 串口控制寄存器 2
0039	0049	0035	0002	RCR1x	McBSP 接收控制寄存器 1
0039	0049	0035	0003	RCR2x	McBSP 接收控制寄存器 2
0039	0049	0035	0004	XCR1x	McBSP 发送控制寄存器 1
0039	0049	0035	0005	XCR2x	McBSP 发送控制寄存器 2
0039	0049	0035	0006	SRGR1x	McBSP 采样率产生寄存器 1
0039	0049	0035	0007	SRGR2x	McBSP 采样率产生寄存器 2
0039	0049	0035	0008	MCR1x	McBSP 多通道寄存器 1
0039	0049	0035	0009	MCR2x	McBSP 多通道寄存器 2
0039	0049	0035	000A	RCERAx	McBSP 接收通道使能寄存器 A
0039	0049	0035	000B	RCERBx	McBSP 接收通道使能寄存器 B
0039	0049	0035	000C	XCERAx	McBSP 发送通道使能寄存器 A
0039	0049	0035	000D	XCERBx	McBSP 发送通道使能寄存器 B
0039	0049	0035	000E	PCRx	McBSP 管脚控制寄存器

因为 VC5402 只有两个 McBSP，所以只有 McBSP0 和 McBSP1。表 2-12 是原文引用 TI 芯片 VC5410 的 McBSP 寄存器的情况，对 VC5402 也适用。其中，RBR[1,2]、RSR[1,2]和 XSR[1,2]是 McBSP 内部专用的数据缓冲寄存器，CPU 和 DMA 都不能直接访问。从图 2-12 可以看出，CPU 或 DMA 对 McBSP 的访问是通过 DRR 和 DXR 寄存器来实现的，在使用 McBSP 时，必须初始化表 2-12 所示的 McBSP 控制寄存器，使 McBSP 工作在正确的状态下。

下面以 SPRC10（McBSP0 串口控制寄存器 1）的初始化为例介绍串口初始化的 C 程序。关于具体写入的控制字内容要根据具体开发项目的需要，请参阅 TI 的技术文档。下面的程序设置 SPRC10 的初始化值为 0x0021。在 C 语言中，十六进制数只能表示为 0x 前缀的形式，而不认别加 h 后缀的形式。例如 0x0001 为正确的表示形式，而 0001h 为错误的十六进制表示形式，下面结合具体程序给出了详细的注解。

```
#define   SPSA0    0x0038
//SPSA0 用作一个指针指向要操作的控制寄存器的位置
#define   SPSD0    0x0039
//写入到 SPSD0 的值会直接写入到 SPSA0 指向的寄存器里
#define   SPRC10   0x0000
//SPRC10 用来定位 SPSA0 这个指针
```

以上为全局地址。

```
ushort   val_sprc10=0x0021;
//val_sprc10 用来定义写入到 SPRC10 控制寄存器内的值
*(short   *)SPSA0=SPRC10;
//将 SPSA0 指向 SPRC10 寄存器
*(short   *)SPSD0=val_sprc10;
//写控制字到 SPRC10 寄存器里
```

通过 DRR 和 DXR 寄存器来实现对 McBSP 的访问。具体的访问方法有两种：一种是 DSP 工作在主动模式下，DSP 去访问 McBSP 控制寄存器 1 和 2 中的 RRDY 和 XRDY 位。当 McBSP 接收到新的数据时，RRDY 位自动置 1；当 XRDY=1 时，表明 McBSP 已准备好可以发送新的数据。当获知 RRDY=1 或 XRDY=1 时，DSP 就会向 McBSP 读取数据或发送新的数据到 McBSP。另一种工作在中断模式下，当 McBSP 有新的数据到来或是准备发送下一个数据时，会向 DSP 发出中断请求，DSP 响应中断并从 McBSP 接收数据或发送一个新的数据到 McBSP 去。在这种方式下，需要在中断向量表中定义相应的中断陷阱。表 2-13 给出了 McBSP 的中断信号，并给出在中断向量表中设置中断陷阱的程序片段。详细的介绍请参考本书第三章“C/C++程序设计”。

下面的程序片段是摘自 vectors.asm，这个文件是以汇编语言的形式给出的，因此其注释用“;”号给出。

```
rint0:      BD    read_MCBSP0     ; 接收中断跳转去执行中断服务程序 read_MCBSP0
            NOP                   ; 空操作
            RETE                  ; 中断返回
```

表 2-13　McBSP 中断信号

Interrupt Name (中断名)	Description (描述)
RINT	Receive interrupt to CPU (向 CPU 发送接收数据中断信号)
XINT	Transmit interrupt to CPU (向 CPU 发送输出数据中断信号)
REVT	Receive synchronization event to DMA (向 DMA 发送接收同步事件)
XEVT	Transmit synchronization event to DMA (向 DMA 发送输出同步事件)
REVTA	Receive synchronization eventA to DMA (向 DMA 发送接收同步事件 A)
XEVTA	Transmit synchronization eventA to DMA (向 DMA 发送输出同步事件 A)

中断服务程序可以用 C 语言编写，与单片机的 C 语言中断程序相类似。关于 McBSP 端口的具体访问的完整程序段将在第三章“C/C++程序设计”中给出，关于 McBSP 和 EHPI8 等硬件抽象高级访问方法将在第四章“DSP/BIOS 程序设计”中讲述。

VC5402 的 McBSP 的特性有：第一，可进行全双工数据通信；第二，双缓冲数据寄存器，允许连续的数据流；第三，端口的收发帧时钟和串口时钟是独立的，可编程设定其极性；第四，McBSP 提供了与 T1/E1 帧、MVIP 和 ST-BUS 兼容设备（MVIP 帧、H.100 帧、SCSA 帧）、IOM-2 兼容设备以及通用的串行接口的直接连接能力；第五，每个串行口有 128 个通道；第六，传送数据格式可以为 8、12、16、20、24 或 32 bit；第七，接收数据时集成了 μ 律和 A 律压扩技术。

7．硬件计时器

VC5402 具有两个带有 4 bit 定标器的 16 bit 的定时电路。这两个定时电路的定时计数器在每过一个 CPU 时钟周期后都会减 1，每当计数器减至 0 时就会产生一个定时中断。可以通过设置特定的控制位来使定时器停止、恢复运行、复位或禁止。

8．时钟发生器

VC5402 的时钟发生器包括一个内部振荡器和一个锁相环（PLL）电路。这个时钟发生器的激励源可以有两种方式：一是外接晶振，它与内部振荡器一起产生参考时钟信号；二是由一个外部时钟源直接驱动。在 DIV 工作模式下，参考时钟源的二分频产生 CPU 时钟；在 PLL 工作模式下，通过对输入时钟信号（X2/CLKIN 脚）倍频到 1 至 31 倍产生 CPU 时钟信号。在 PLL 模式下，由时钟模式寄存器（CLKMD）来定义倍频数。在上电硬复位时，CLKMD 寄存器的值由外部管脚 CLKMD1、CLKMD2 和 CLKMD3 的电平来决定，复位时的 CLKMD 设置如表 2-14 所示。

完成复位后，再通过设置 CLKMD 的值使 VC5402 工作在需要的时钟频率下。

表 2-14 复位时 CLKMD 寄存器的值及含义

CLKMD1	CLKMD2	CLKMD3	CLKMD RESET VALUE (复位值)	CLOCK MODE (时钟工作模式)
0	0	0	E007h	PLL×15
0	0	1	9007h	PLL×10
0	1	0	4007h	PLL×5
1	0	0	1007h	PLL×2
1	1	0	F007h	PLL×1
1	1	1	0000h	1/2(PLL disabled)(关闭 PLL 模式)
1	0	1	F000h	4/1(PLL disabled)(关闭 PLL 模式)
0	1	1	—	Reserved (bypass mode) (保留(忽略 CLKMD 设置))

2.2.4 寄存器与中断

VC5402 具有 27 个 CPU 存储器映射寄存器和 33 个片上外设存储器映射寄存器（包括 4 个 McBSP 和 DMA 的子地址存储器映射寄存器）。为了避免与前文重复，在此仅列出了 CPU 和片上外设的存储器映射寄存器，如表 2-15 和表 2-16 所示。

表 2-15 CPU 存储器映射寄存器

名称	地址		含义	名称	地址		含义
	十进制表示	十六进制表示			十进制表示	十六进制表示	
IMR	0	0	中断屏蔽寄存器	AR2	18	12	辅助寄存器 2
IFR	1	1	中断标志寄存器	AR3	19	13	辅助寄存器 3
-	2-5	2-5	测试保留位域	AR4	20	14	辅助寄存器 4
ST0	6	6	状态寄存器 0	AR5	21	15	辅助寄存器 5
ST1	7	7	状态寄存器 1	AR6	22	16	辅助寄存器 6
AL	8	8	累加器 A 低字(15-0)	AR7	23	17	辅助寄存器 7
AH	9	9	累加器 A 高字(31-16)	SP	24	18	堆栈指针寄存器
AG	10	A	累加器 A 保护字(39-32)	BK	25	19	循环块大小寄存器
BL	11	B	累加器 B 低字(15-0)	BRC	26	1A	块循环指针寄存器
BH	12	C	累加器 B 高字(31-16)	RSA	27	1B	块循环起始地址寄存器
BG	13	D	累加器 B 保护字(39-32)	REA	28	1C	块循环终止地址寄存器
TREG	14	E	暂存器	PMST	29	1D	微处理器模式状态寄存器
TRN	15	F	过渡态寄存器	XPC	30	1E	扩展程序页指针
AR0	16	10	辅助寄存器 0	-	31	1F	保留
AR1	17	11	辅助寄存器 1				

表 2-16　片上外设存储器映射寄存器

寄存器名	地址 (十六进制)	描　　述	类　属
DRR20	20	McBSP0 数据接收寄存器 2	McBSP #0
DRR10	21	McBSP0 数据接收寄存器 1	McBSP #0
DXR20	22	McBSP0 数据发送寄存器 2	McBSP #0
DXR10	23	McBSP0 数据发送寄存器 1	McBSP #0
TIM	24	计时器 0 寄存器	Timer0
PRD	25	计时器 0 周期计数器	Timer0
TCR	26	计时器 0 控制寄存器	Timer0
-	27	保留	
SWWSR	28	软件等待状态寄存器	扩展总线
BSCR	29	块开关控制寄存器	扩展总线
-	2A	保留	
SWCR	2B	软件等待状态控制寄存器	扩展总线
HPIC	2C	HPI 控制寄存器	HPI
-	2D - 2F	保留	
TIM1	30	计时器 1 寄存器	Timer1
PRD1	31	计时器 1 周期计数器	Timer1
TCR1	32	计时器 1 控制寄存器	Timer1
-	33 - 37	保留	
SPSA0	38	McBSP0 子段地址寄存器	McBSP #0
SPSD0	39	McBSP0 子段数据寄存器	McBSP #0
-	3A – 3B	保留	
GPIOCR	3C	通用 I/O 管脚控制寄存器	GPIO
GPIOSR	3D	通用 I/O 管脚状态寄存器	GPIO
-	3E – 3F	保留	
DRR21	40	McBSP1 数据接收寄存器 2	McBSP #1
DRR11	41	McBSP1 数据接收寄存器 1	McBSP #1
DXR21	42	McBSP1 数据发送寄存器 2	McBSP #1
DXR11	43	McBSP1 数据发送寄存器 1	McBSP #1
-	44 – 47	保留	
SPSA1	48	McBSP1 子段地址寄存器	McBSP #1
SPSD1	49	McBSP1 子段数据寄存器	McBSP #1
-	4A – 53	保留	
DMPREC	54	DMA 通道优先级和使能控制寄存器	DMA
DMSA	55	DMA 子段地址寄存器	DMA
DMSDI	56	DMA 子段数据寄存器（自增模式）	DMA
DMSDN	57	DMA 子段数据寄存器	DMA
CLKMD	58	时钟工作模式寄存器	PLL
-	59 – 5F	保留	

VC5402 所有内部和外部中断的中断号与优先级列于表 2-17 中。

表 2-17　VC5402 的中断

中 断 名	相对位置		优先级	功　　能
	十进制数表示	十六进制数表示		
$\overline{\text{RS}}$，SINTR	0	00	1	硬复位和软复位
$\overline{\text{NMI}}$, SINT16	4	04	2	非屏蔽中断
SINT17	8	08	-	软中断 17 号
SINT18	12	0C	-	软中断 18 号
SINT19	16	10	-	软中断 19 号
SINT20	20	14	-	软中断 20 号
SINT21	24	18	-	软中断 21 号
SINT22	28	1C	-	软中断 22 号
SINT23	32	20	-	软中断 23 号
SINT24	36	24	-	软中断 24 号
SINT25	40	28	-	软中断 25 号
SINT26	44	2C	-	软中断 26 号
SINT27	48	30	-	软中断 27 号
SINT28	52	34	-	软中断 28 号
SINT29	56	38	-	软中断 29 号
SINT30	60	3C	-	软中断 30 号
$\overline{\text{INT0}}$, SINT0	64	40	3	外部中断 0
$\overline{\text{INT1}}$, SINT1	68	44	4	外部中断 1
$\overline{\text{INT2}}$, SINT2	72	48	5	外部中断 2
TINT0, SINT3	76	4C	6	计时器 0 中断
BRINT0, SINT4	80	50	7	McBSP0 接收数据中断
BXINT0, SINT5	84	54	8	McBSP0 发送数据中断
保留（DMAC0），SINT6	88	58	9	默认状态下保留；也可以通过设置 DMPREC 寄存器，将其作为 DMA 通道 0 的中断
TINT1(DMAC1), SINT7	92	5C	10	默认状态下为计时器 1 中断；也可以通过设置 DMPREC 寄存器，将其作为 DMA 通道 1 的中断
$\overline{\text{INT3}}$, SINT8	96	60	11	外部中断 3
HPINT, SINT9	100	64	12	HPI 中断
BRINT1(DMAC2), SINT10	104	68	13	默认状态下为 McBSP1 接收中断；也可以通过设置 DMPREC 寄存器，将其作为 DMA 通道 2 的中断
BXINT1(DMAC3), SINT11	108	6C	14	默认状态下为 McBSP1 发送中断；也可以通过设置 DMPREC 寄存器，将其作为 DMA 通道 3 的中断
DMAC4, SINT12	112	70	15	DMA 通道 4 中断
DMAC5, SINT13	116	74	16	DMA 通道 5 中断
保留	120 – 127	78 – 7F	-	保留

表 2-17 中，NMI 是一个不可被中断屏蔽寄存器屏蔽的外部中断。所有表 2-17 中的这些中断信号需要在中断向量表程序（vectors.asm）中给出，当需要 DSP 响应这些中断信号时，就在相应的中断名后面加上跳转程序。中断向量表程序最好用汇编语言编写，其格式比较固定，可以参考第 2.4 节“SY-5402EVM 板”。在 DSP/BIOS 环境下有专用的 C 语句格式的中断调用方法，请参考第四章“DSP/BIOS 程序设计”。其中，RS 中断指的是复位信号，它具有最高的优先级，跳转地址一般设为 c_int00，即 C 程序的入口地址。当多个中断同时发生时，DSP 首先响应中断优先级别高的中断信号。

VC5402 的中断标志寄存器（IFR）和中断屏蔽寄存器（IMR）及其各位的含义如图 2-13 和表 2-18 所示。

15-14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES	DMAC5	DMAC4	BXINT or DMAC3	BRINT or DMAC2	HPIN	INT3	TINT1 or DMAC1	RES or DMAC0	BXINT	BRINT	TINT	INT2	INT1	INT0

图 2-13　IFR 和 IMR 中断寄存器

表 2-18　IFR 和 IMR 寄存器各位的含义

位		功　　能
位号	名　称	
15 – 14	-	为以后的功能扩展备用
13	DMAC5	DMA 通道 5 中断标志/屏蔽位
12	DMAC4	DMA 通道 4 中断标志/屏蔽位
11	BXINT1/DMAC3	这一位可以作为 McBSP1 发送数据中断标志/屏蔽位，也可以作为 DMA 通道 3 中断标志/屏蔽位。通过设置 DMPREC 寄存器做出上述选择
10	BRINT1/DMAC2	这一位可以作为 McBSP1 接收数据中断标志/屏蔽位，也可以作为 DMA 通道 2 中断标志/屏蔽位。通过设置 DMPREC 寄存器做出上述选择
9	HPINT	主机中断 DSP 的中断标志/屏蔽位
8	INT3	外部中断 3 的中断标志/屏蔽位
7	TINT1/DMAC1	这一位可以作为计时器 1 中断标志/屏蔽位，也可以作为 DMA 通道 1 中断标志/屏蔽位。通过设置 DMPREC 寄存器做出上述选择
6	保留/DMAC0	这一位可以保留，也可以作为 DMA 通道 0 中断标志/屏蔽位。通过设置 DMPREC 寄存器做出上述选择
5	BXINT0	McBSP0 发送数据中断标志/屏蔽位
4	BRINT0	McBSP0 接收数据中断标志/屏蔽位
3	TINT0	计时器 0 中断标志/屏蔽位
2	INT2	外部中断 2 的中断标志/屏蔽位
1	INT1	外部中断 1 的中断标志/屏蔽位
0	INT0	外部中断 0 的中断标志/屏蔽位

通过设置中断屏蔽寄存器 IMR 的值来开启（或“使能”）或屏蔽相应的可屏蔽中断，通过查看中断标志寄存器 IFR 的值可以了解哪些中断是开启（或“使能”）的，哪些中断是被屏蔽掉的。这里的“使能”是指哪些中断是处于可响应状态或是处于可激活状态，但 DSP 并没有响应该中断，它的含义与“屏蔽”相反。

2.3 TMS320VC5510 简介

C55x 系列 DSP 具有相同的 CPU 结构，与 C54x 一样是低功耗、高性能的定点 DSP 系列。下面仅以 TMS320VC5510（以下简称 VC5510）为例简单地介绍它们的 CPU、存储器和片上外设。

2.3.1 CPU

VC5510 支持改进的哈佛总线结构，包括一条程序总线、三条数据读总线、两条数据写总线和附加的片上外设以及 DMA 专用总线。这种总线结构可以使得 VC5510 在一个时钟周期内完成三个数据读操作和两个数据写操作，DMA 可以在无需 CPU 干预的情况下在一个时钟周期内完成两个数据的传输。

VC5510 提供了两个乘加器（MAC）单元，每个 MAC 均可在一个指令周期内完成一个 17×17 bit 的乘加运算。VC5510 具有一个 40 bit 的算术逻辑单元（ALU）和一个附加的 16 bit 算术逻辑单元（ALU），这两个 ALU 可以优化 CPU 的并行处理并起到节约功耗的作用。

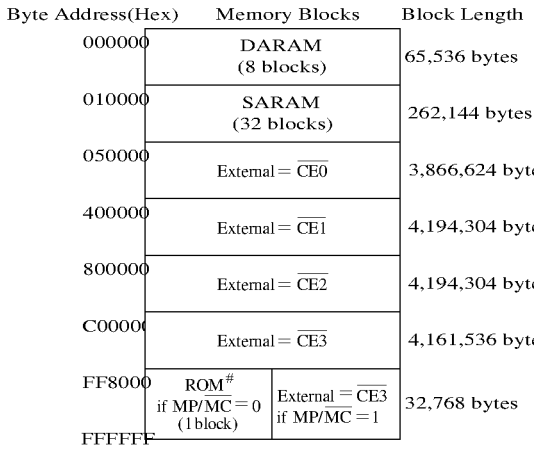
VC5510 支持可变长度指令集。

VC5510 的片上外设包括一个扩展的存储器接口（EMIF），通过 EMIF 可以对外部异步存储器（如 EPROM 和 SARAM 等）和内部同步存储器（如 DRAM 和 SARAM 等）进行无缝访问。

VC5510 有两种型号，CPU 时钟速率分别为 160 MHz 和 200 MHz。

2.3.2 存储器配置

VC5510 片上的物理存储器包括一个 64 K×16 bit 的 DARAM（分成了等容量的 8 块）、一个 256 K×16 bit 的 SARAM（分成了等容量的 32 块）和一个 32 K×16 bit 的 ROM，一共具有 352 K×16 bit 的存储空间。其中，DARAM 和 SARAM 可以被程序总线、数据总线和 DMA 总线访问，在 ROM 中固化了 Boot Loader 程序、256 个值的正弦查找表（360° 的 256 等分间隔）和向量表等。在 C54x 中数据映射存储空间是固定的 64 K×16 bit ，一般不能进行扩展。在 C55x 中，数据映射存储空间是和程序映射存储空间同在一块映射存储块中的，即地址是统一分布的，从映射存储空间中不能分清数据空间和程序空间。于是 VC5510 的程序区地址和数据区地址是不能重合的，且数据区可以扩展到 64 K×16 bit 以上的空间。图 2-14 是 VC5510 的映射存储器示意图，从图上可以看出 DARAM 被固定地映射到 000000h 至 00FFFFh 处，SARAM 被固定地映射到 010000h 至 04FFFF 处，ROM 被映射到 FF8000h 至 FFFFFFh 处。当 MP/MC=1 时，外部的存储器被映射到相应的位置上，否则，这些空间是不能用来存储程序和数据的。



(1) 图中地址仅给出了每块存储区的首地址。

(2) 双访问 RAM（DARAM）：每个时钟周期可以访问二次，共分为 8 块，每块 8 KB。

(3) 单访问 RAM（SARAM）：每个时钟周期仅能访问一次，共分为 32 块，每块 8 KB。

(4) 通过管脚 CE[0:3]可以选中访问外部存储空间，支持异步存储器、同步动态存储器(SDRAM)和同步突发静态存储器（SBSRAM）。

(5) 只读存储器（ROM）：每两个时钟周期访问一次，只有一块，大小为 32 KB。

图 2-14　VC5510 映射存储器

2.3.3　片上外设

VC5510 具有以下片上外设：

(1) 外部存储器接口（EMIF）；

(2) 6 个通道的 DMA 控制器；

(3) 16 bit 的并行增强主机接口（EHPI16）；

(4) 一个数字锁相环（DPLL）时钟发生器；

(5) 两个硬件定时器；

(6) 三个多通道缓冲串行口（McBSPs）；

(7) 8 个可配置的通用 I/O 管脚。

本书不再详细介绍 VC5510 的各种片上外设，如有需要，请参阅相关的 TI 技术资料。

2.4　SY-5402EVM 板

2.4.1　SY-5402EVM 板的硬件组成

SY-5402EVM 是由上海三意电子公司设计制造的 DSP 功能板。

EVM 板不仅是学习 DSP 程序设计方法的有利硬件平台，也可以作为开发人员开发实际 DSP 应用系统的最好的参考平台。EVM 板的作用表现在很多方面：① 对于初学 DSP 程序设计者，EVM 板提供了一整套直观意义上的 DSP 工作硬件平台，可帮助初学者迅速熟悉开发的整个硬件环境，能较容易地开展面向实际 DSP 系统的程序开发。本书仅就 SY-5402EVM 板向初学者展开其硬件组成，使初学者可以直观地理解 DSP 应用系统的硬件结构。② 对于软件工程师或是数字信号算法研究人员，EVM 板为其提供了一个相当有效的测试环境，软

件工程师可以直接在 EVM 板上测试其算法的可行性。这个测试过程原来可能只限于在 MATLAB 或其他数学软件上完成，而借助于通用数学软件离开发项目的评估距离太远，不如借助 EVM 板更加具有直接的意义。也就是说，EVM 板在一定意义上可以节约软件设计人员的软件开发时间。在 EVM 板上可行的程序，特别是 C/C++程序，具有较好的可移植性，可以直接移植到开发的项目中去。③ 对于硬件设计师而言，EVM 板也有较强的针对性。购买 EVM 板都能取得详细的硬件结构原理图，这些原理图是硬件设计师开发 DSP 应用系统最好的参考资料。可以在 EVM 板原理图的指导下，迅速完成设计开发项目的硬件系统原理图。一般来说，DSP 及其与外设的连接是原理图中最复杂的核心部分，这部分原理图的完成，开发项目的硬件设计工作也就完成了大部分。当然，也可以直接将 EVM 板作为开发项目硬件结构的一部分应用到开发项目中去。

SY-5402EVM 板的核心处理器是 VC5402，除了具有上述通用 EVM 板的优点之外，它还具有以下的特性：

(1) 外部单+5 V 电源供电。

(2) 提供了与计算机串口的数据通信功能，可以与计算机交换数据。

(3) 提供了语音输入和语音输出的接口，可以进行语音信号的分析与处理。

(4) 提供了可与外部相连接的 EHPI8、并口、McBSP 口等 VC5402 片上外设接口。通过这些接口，可以方便地对 SY-5402EVM 板进行功能扩展。

(5) SY-5402EVM 板上提供了大容量的外部程序和数据 RAM 以及 FLASH 存储空间，方便了用户开发和使用。

(6) SY-5402EVM 板集成了一块 ALTRA 公司的 CPLD，用户可以通过该 CPLD 实现对 EVM 板上各种逻辑的控制管理。

(7) SY-5402EVM 板提供了必要的复位、测试针和开关插销，为用户调试和测量提供了方便。

(8) SY-5402EVM 板集成度相当高，板面积很小，易于现场测试应用。

本书中仅用到了 SY-5402EVM 板语音处理部分的一部分硬件，完成了输入语音信号经过功放、模数变换、自适应增益控制（AGC）、数模变换、电压调整和输出语音信号的功能，详见第七章“一个完整实例”。下面将对这一部分的硬件电路及其实现的功能作一个详细的介绍。

实现上述语言处理功能用到了 VC5402 的串行口 1（McBSP1），这些口线接到了 CPLD 的 I/O 口线上，然后，A/D/A 与功放和输入、输出语音端子相连接，如图 2-15 和图 2-16 所示(这里没有列出电源芯片和时钟电路)。这部分功能的实现仅使用了 TMS320VC5402、EPM7128SLC84-15、TLC320AD50C、TLC274 以及各种电阻、电容、接插件、电源芯片和 FLASH 等。从这一小部分可以看到，SY-5402EVM 板的一个重大特色就在于尽可能地使用了 TI 公司的各种芯片。关于 SY-5402EVM 板的其他情况请登陆三意电子公司的网站 www.dspdsp.com。

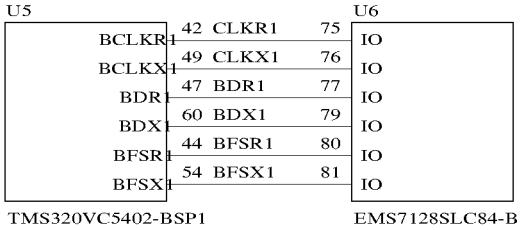


图 2-15 VC5402 的串行口 1 和 CPLD 的 I/O 口线的连接情况

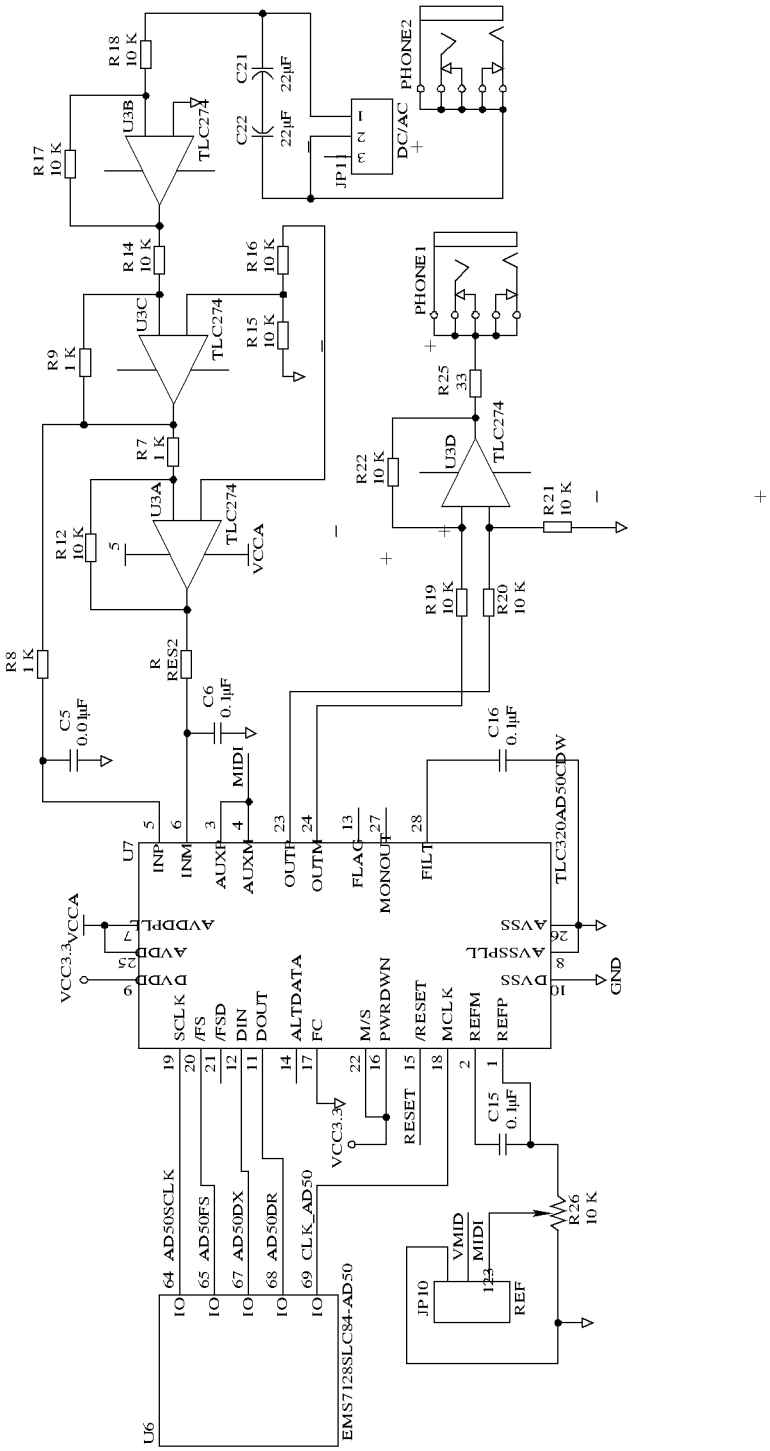


图 2-16　语音信号与 A/D/A 间的连接情况

2.4.2　VC5402 的存储器配置

1. VC5402 的存储器配置文件

本书所有给出的 C/C++程序均在三意电子公司的 SY-5402EVM 板上调试通过，程序中使用了下面的存储器配置文件（user_learn.cmd）。下面给出这个配置文件的全部源码，并给出详细的解释。文件中的注释需采用 C 语言风格，即用“/*　*/”可以注释多行或一行，特别注意：不能用“//”来注释一行。然后，在第 2.4.2 节的第 2 个问题中给出了如何编写存储器配置文件。参考图 2-1，可以对这个文件有更好的理解。

```
/*filename: user_learn.cmd
*This file is a configuration file for VC5402 Prgram and Data Space
*Last modified by Zhang Yong.       */
-m    user_learn.map
MEMORY
{
      PAGE 0:    PROG(RWXI):   origin = 0x80,     length = 0x3000
                 VECS(RXI):    origin=0xFF80,  length =0x80
      PAGE 1:    DATA(RWI):    origin = 0x3080,  length = 0xF80
}
/*  MEMORY 为存储器指示关键字，程序区用 PAGE 0 来表示，数据区用 PAGE 1 来表示。程序区和数据区都使用了 VC5402 片上的 16 K×16 bit 的 DARAM。程序区和数据区的名称可以按 C 语言命名规则任意取名。程序区命名为 PROG，占据了约 12 K×16 bit 的空间；数据区命名为 DATA，占据了约 4 K×16 bit 的 DARAM 空间。origin = 0x80 代表了程序空间的起始地址，len = 0x3000 代表程序空间的长度为 0x3000。
*/
SECTIONS
{
          .text       {}  >  PROG   PAGE 0
          .cinit      {}  >  PROG   PAGE 0
          .pinit      {}  >  PROG   PAGE 0
          .vectors    {}  >  VECS   PAGE 0
          .stack      {}  >  DATA   PAGE 1
          .bss:       {}  >  DATA   PAGE 1
          .const:     {}  >  DATA   PAGE 1
          .switch:    {}  >  DATA   PAGE 1
          .sysmem:    {}  >  DATA   PAGE 1
          .cio:       {}  >  DATA   PAGE 1
}
```

/*　SECTIONS 是存储器区段关键字；.text 等是汇编指示标志关键字，它的含义为将程序段放入 PAGE 0 的 PROG 区域；.cinit 指示将 C 语言初始化数据存入 PROG 区域中；.pint 指示将 C++程序初始化数据存入 PROG 区域；.stack 表明在 PAGE 1 的 DATA 区域开辟堆栈；.bss 指示将未初始化的变量分配在 DATA 区域；.const 指示将常量分配在 DATA 区域；.switch 指示将 C 语言的 switch 语句分配到 DATA 区域；.sysmem 指示将 C 语句（如 malloc 或 alloc）开辟的存储区分配到 DATA 区域。在 C/C++程序设计中，主要用到了这些汇编连接存储空间分配指示标志，即只有在连接时才需要进行存储器配置，这时才需要.cmd 文件。

对于不同型号的 DSP 芯片，有不同的存储器配置方案，即使对于同一种 DSP 芯片，使用在不同的开发项目中，其存储器配置方案也不尽相同。但要遵循一条基本的原则，就是最大可能地、合理地使用 DSP 芯片的片上存储资源，减少浪费。

在.cmd 文件中，可以加入连接指令。最常用的是-m 指令，　用法如 user_learn.cmd 中的“-m user_learn.map”，这个命令用来生成一个连接 MAP 表，文件名为 user_learn.map。这个 MAP 表是开发 DSP 程序中很有用的一个文件，它向软件设计人员详细展示了存储器的配置和使用情况、各个段的绝对地址和重新分配后的全局符号。下面是作者的一个程序生成的 MAP 表文件，文件名为 user_learn.map，完整的全文如下：

```
******************************************************************************
TMS320C54x COFF Linker                  PC Version 3.70
******************************************************************************
>> Linked Thu Dec 12 18:30:02 2002

OUTPUT FILE NAME:    <./Release/ifddc.out>
ENTRY POINT SYMBOL: "_c_int00"    address: 00000080

MEMORY CONFIGURATION
            name            origin          length          used            attr    fill
            ----------------  --------        ---------       --------        ----    --------
PAGE  0: PROG               00000080        00003000        000005a6        RWIX
            VCTS            0000ff80        00000080        00000064        R IX

PAGE  1: DATA               00003080        00000f80        000008e6        RWI

SECTION ALLOCATION MAP

output                                                  attributes/
section     page    origin          length          input sections
--------    ----    ---------       ---------       ----------------
.text       0       00000080        0000037f
```

```
                    00000080    0000004a    rts.lib : boot.obj (.text)
                    000000ca    00000000    vectors.obj (.text)
                    000000ca    00000054    rts.lib : exit.obj (.text)
                    0000011e    00000007              : _lock.obj (.text)
                    00000125    000000b2    ifddc.obj (.text)
                    000001d7    0000009d    rts.lib : f_add.obj (.text)
                    00000274    00000020              : f_cmp.obj (.text)
                    00000294    0000009a              : f_div.obj (.text)
                    0000032e    00000001              : f_error.obj (.text)
                    0000032f    00000073              : f_mul.obj (.text)
                    000003a2    00000009              : f_sub.obj (.text)
                    000003ab    00000054    54xdsp.lib : fltoq15.obj (.text)

.cinit      0       000003ff    00000227
                    000003ff    00000009    rts.lib : exit.obj (.cinit)
                    00000408    00000006              : _lock.obj (.cinit)
                    0000040e    00000217    ifddc.obj (.cinit)
                    00000625    00000001    --HOLE-- [fill = 0000]

.pinit      0       00000080    00000000

.vectors    0       0000ff80    00000064    //一般要重新定位到程序空间中
                    0000ff80    00000064    vectors.obj (.vectors)

.stack      1       00003080    00000400    UNINITIALIZED
                    00003080    00000000    rts.lib : boot.obj (.stack)

.bss        1       00003480    000004e2    UNINITIALIZED
                    00003480    000004bd    ifddc.obj (.bss)

                    0000393d    00000000    rts.lib : f_add.obj (.bss)
                    0000393d    00000000              : f_cmp.obj (.bss)
                    0000393d    00000000              : boot.obj (.bss)
                    0000393d    00000000    vectors.obj (.bss)
                    0000393d    00000000    rts.lib : f_div.obj (.bss)
                    0000393d    00000000              : f_sub.obj (.bss)
                    0000393d    00000000    54xdsp.lib : fltoq15.obj (.bss)
                    0000393d    00000000    rts.lib : f_error.obj (.bss)
```

```
                    0000393d    00000000        : f_mul.obj (.bss)
                    0000393d    00000023        : exit.obj (.bss)
                    00003960    00000002        : _lock.obj (.bss)

.const      1       00003962    00000004
                    00003962    00000004        ifddc.obj (.const)

.switch     1       00003080    00000000        UNINITIALIZED

.sysmem     1       00003080    00000000        UNINITIALIZED

.cio        1       00003080    00000000        UNINITIALIZED

.data       1       00000000    00000000        UNINITIALIZED
                    00000000    00000000        rts.lib : boot.obj (.data)
                    00000000    00000000        54xdsp.lib : fltoq15.obj (.data)
                    00000000    00000000        rts.lib : f_sub.obj (.data)
                    00000000    00000000              : f_mul.obj (.data)
                    00000000    00000000              : f_error.obj (.data)
                    00000000    00000000              : f_div.obj (.data)
                    00000000    00000000              : f_cmp.obj (.data)
                    00000000    00000000              : f_add.obj (.data)
                    00000000    00000000        vectors.obj (.data)
                    00000000    00000000        ifddc.obj (.data)
                    00000000    00000000        rts.lib : _lock.obj (.data)
                    00000000    00000000              : exit.obj (.data)
```

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name //按 ASCII 顺序排列

```
address         name
--------        ----
00003480        .bss
00000000        .data
00000080        .text
0000011b        C$$EXIT
000001d7        F$$ADD
00000274        F$$COMPARE
00000294        F$$DIV
```

0000032f　F$$MUL
000003a2　F$$SUB
00003677　_BB_data
0000393a　_DMSA
0000393b　_DMSDN
0000393c　_DMSRC5
000001d6　_DataIO
00003480　_I_data_in
0000357a　_Q_data_in
00000400　_STACK_SIZE
00003480　_bss_
000003ff　_cinit_
00000000　_data_
00000000　_edata_
00003962　_end_
000003ff　_etext_
ffffffff　_pinit_
00000080　_text_
0000395e　_cleanup_ptr
0000395f　_dtors_ptr
00000001　_lflags
00003960　_lock
0000011e　_nop
0000011f　_register_lock
00000122　_register_unlock
00003961　_unlock
00003936　_a1
0000011b　_abort
000000f9　_atexit
000037a8　_bbpart
00000080　_c_int00
00003674　_drr10
000000ca　_exit
0000032e　_f$$error
00003830　_fbbpart
00003828　_fipart
000003ab　_fltoq15
0000382c　_fqpart

```
00003930    _h
00003934    _i
000037a4    _ipart
00000133    _main
00000125    _myISP
00003938    _p1
00003939    _p2
000037a6    _qpart
00003676    _spcr10
00003675    _spsa0
000036dc    _tempt
000003ff    cinit
00000000    edata
00003962    end
000003ff    etext
ffffffff    pinit

GLOBAL SYMBOLS: SORTED BY Symbol Address     //按符号地址顺序排列

address     name
--------    ----
00000000    _data_
00000000    _edata_
00000000  . data
00000000    edata
00000001    _lflags
00000080    _text_
00000080    .text
00000080    _c_int00
000000ca    _exit
000000f9    _atexit
0000011b    C$$EXIT
0000011b    _abort
0000011e    _nop
0000011f    _register_lock
00000122    _register_unlock
00000125    _myISP
00000133    _main
```

000001d6　_DataIO
000001d7　F$$ADD
00000274　F$$COMPARE
00000294　F$$DIV
0000032e　_f$$error
0000032f　F$$MUL
000003a2　F$$SUB
000003ab　_fltoq15
000003ff　etext
000003ff　_cinit_
000003ff　_etext_
000003ff　cinit
00000400　_STACK_SIZE
00003480　_I_data_in
00003480　_bss_
00003480　.bss
0000357a　_Q_data_in
00003674　_drr10
00003675　_spsa0
00003676　_spcr10
00003677　_BB_data
000036dc　_tempt
000037a4　_ipart
000037a6　_qpart
000037a8　_bbpart
00003828　_fipart
0000382c　_fqpart
00003830　_fbbpart
00003930　_h
00003934　_i
00003936　_a1
00003938　_p1
00003939　_p2
0000393a　_DMSA
0000393b　_DMSDN
0000393c　_DMSRC5
0000395e　_cleanup_ptr
0000395f　_dtors_ptr

```
00003960        _lock
00003961        _unlock
00003962        end
00003962        _end_
ffffffff        pinit
ffffffff        _pinit_

[61 symbols]
```

以上为 MAP 表文件内部的内容，所有 DSP 程序开发最后都不可避免地要查看 MAP 表，对程序各个段和符号的定位有一个明确的核实。这样做有助于帮助理解程序的运行段、装入段及其资源的利用情况。所以在此作者给出一个详细的、没有经过删节的 MAP 表文件来说明这个 MAP 表的重要性，供读者参考。本 MAP 表的原程序在 SY-5402EVM 板上正常运行。从上面可以看出 MAP 表的格式比较固定，不再详细解释。

2．如何编写存储器配置文件

面向 DSP 的 C/C++程序都要经过编译、汇编和连接成可执行目标文件后，才能在 DSP 芯片上运行。编译、汇编和连接的意义与 ANSI-C 是相同的。编译器（Compiler）将 C/C++源文件编译成汇编语言代码，这一步只对采用 C/C++编写的程序才有必要。如果是采用汇编语言编写程序，则不会调用编译器。在编译过程中一个重要的方面就是编译器集成了一个对 C/C++代码的优化器。这个优化分为几个级别：对采用 DSPLIB 库的 C/C++代码优化率为 100%，对于模块化结构化的 C/C++程序的优化效率可达到 80%以上。汇编器（Assembler）将汇编语言代码汇编为机器语言代码，这种代码具有通用目标语言（COFF）的格式。连接器（Linker）将各个文件形成的目标文件以及运行支撑库连接成一个可执行的目标文件。在这个连接过程中，连接器还会将存储空间进行分段，并将相应的程序代码与这个区段的绝对地址进行绑定，即将目标文件分配到各个存储区段内。完成这个工作的语言称为连接命令语言，文件的扩展默认为（.cmd）。图 2-17 为 C/C++程序编译连接流程框图（原文引自 TI 资料库）。

连接命令语言是 ASCII 码形式的文件。它支持 C 语言的“/*　*/”注释，不支持 C++的“//”注释说明。连接命令文件中可以写入连接命令，但在 CCStudio 集成环境中无疑是画蛇添足。这个文件的总体框架如下：

```
MEMORY
{    /* 存储器配置 */
     PAGE  0:
     PAGE  1:
}
SECTIONS
{
}
```

其中，MEMORY 是连接命令语言的关键字，它的内容指示 DSP 映射存储器的配置。

一般包括两个页面，即 PAGE 0 和 PAGE 1，PAGE 0 指程序存储空间，PAGE 1 指数据存储空间，最多可以设置到第 255 个页面；SECIONS 也是关键字，它的内容是具体指示程序中各个段在 MEMORY 中的分布情况。

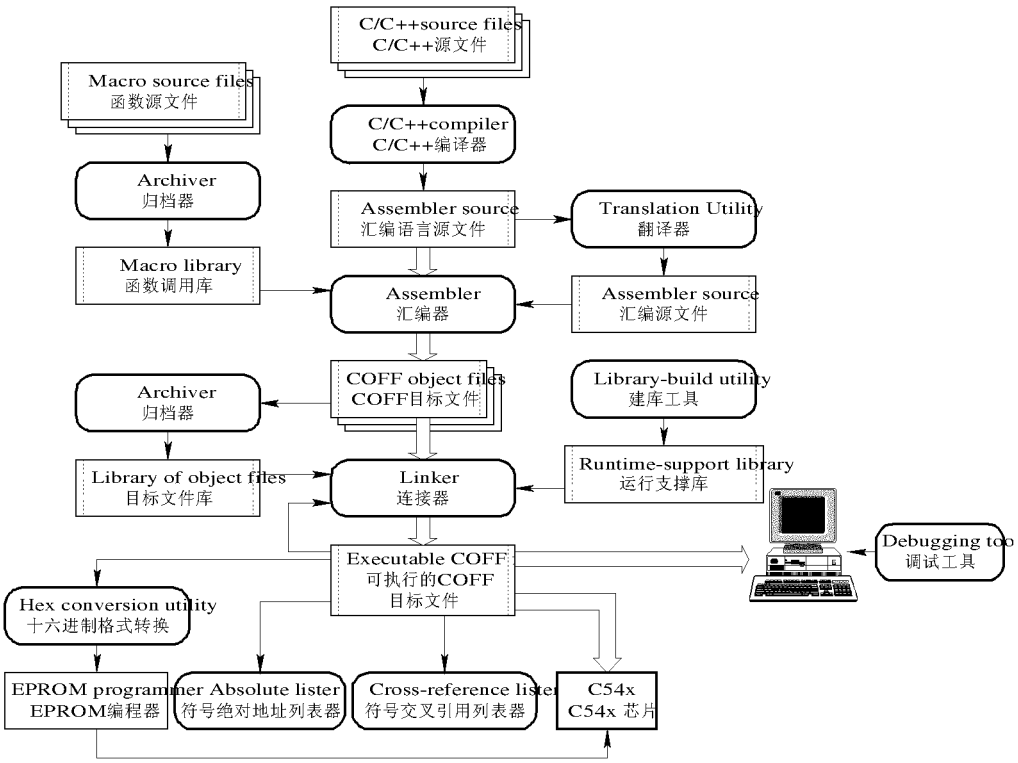


图 2-17　C/C++语言程序编译连接流程框图

现将连接命令语言的所有关键字列在下面：

align(或 ALIGN)　attr(或 ATTR)　block(或 BLOCK)　COPY

DSECT　f(或 fill 或 FILL)　group(或 GROUP)　load(或 LOAD 或>)

l (或 len 或 length 或 LENGTH)　MEMORY　NOLOAD　page(或 PAGE)

o(或 org 或 origin 或 ORIGIN)　range　run(或 RUN)　SECTIONS

spare　type(或 TYPE)　UNION

用在连接命令文件中的常数可以采用汇编语言格式，也可以采用 C 语言格式，还可以混用。例如，对于常数 100（十进制），用汇编语言表示为 100（十进制）、144q（八进制）、64h（十六进制）；用 C 语言表示形式为 100（十进制）、0144（八进制，注意前缀为零而不是字母 O）、0x144（十六进制）。

实际编写连接命令文件时，常用的关键字并不多。使用这些常用的连接命令语言的关键字将 user_learn.cmd 文件改写，结合这个改写的文件具体说明这些常用关键字的用法。

下面为改写后的 user_learn.cmd 文件，此文件在 SY-5402EVM 板上测试通过，在此仅作为示例。

```
MEMORY
{
    PAGE 0:    PROG(RWXI):  o=0x80,      l=0x3000
    PAGE 1:    DATA(RWI):   o=0x3080,    l=0xF00
    PAGE 2:    ONCHIP(RW):  o=0x3F00,    l=0x80,   f=0x0000
}
/*  MEMORY 指示符的通用语法为：
     MEMORY
     {
          PAGE 0:   name0[(attr)]: origin=constant , length=constant
          PAGE n:   namen[(attr)]: origin=constant , length=constant
     }
```

其中，attr 可以为 W、R、X、I 的组合形式，其中的 W 代表此存储区段可写，R 代表可读，X 代表可存入运行代码，I 代表可以初始化。本例 MEMORY 中定义了三个页面，分别命名为 PROG、DATA 和 ONCHIP。ONCHIP 中没有定义段，所以初始化填充了 0x0000（填充必须是两个字节）。

```
*/
SECTIONS
{
     .cinit:     {} > PSARAM1   PAGE 0
     .pinit:     {} > PSARAM1   PAGE 0
     .text       load=PROG,  run=0x2000,    align=0x2000
     .cinit      {} > PROG PAGE 0,     align=0x80
     .pint       >PROG     align   0x20     PAGE 0
     .vectors    >PROG   PAGE 0
     .stack      {} > DATA PAGE 1
     .bss:       {} > DATA PAGE 1
     .const:     {} > DATA PAGE 1
     .switch:    {} > DATA PAGE 1
     .sysmem:    {} > DATA PAGE 1
}
/* SECTIONS 的通用语法为：
     SECTIONS
     {
          name: [property, property, property, …]
```

```
        name: [property, property, property, …]
        name: [property, property, property, …]
    }
```

其中，load 指示此段在存储器中装入的位置，run 指示此段在存储器中运行的位置，align 为此段分配具体的空间。{}中可以标明该段所在的目标文件。

*/

当在程序中不加入存储器配置文件（.cmd）时，连接器会使用默认的.cmd 文件。这种方法虽然方便，但不利于开发初期程序的调试和查错。

2.4.3　VC5402 的中断向量表

C/C++程序中需要响应中断处理时，应将中断向量表文件加入到工程中去。这个文件的格式也比较固定，下面给出一个在 SY-5402EVM 板测试成功的向量表文件。请参考表 2-17 阅读下列文件，该文件是用汇编语言编写的。

```
;Filename:  vectors.asm
        .sect ".vectors"
        .ref _myISP             ; 主程序中的一个 C 中断响应函数
        .ref _c_int00           ;  C 程序执行入口
        .align  128
RESET:                          ; 复位中断，优先级最高
        BD      _c_int00        ; 执行完该语句下面的一条语句后，跳转到 c_int00 处
        STM     #128, SP        ; 设置堆栈大小为 128 个字
nmi:    RETE                    ; nmi 非屏蔽中断返回
                NOP
                NOP
                NOP
                                ; VC5402 的软中断
sint17  .space  4*16
sint18  .space  4*16
sint19  .space  4*16
sint20  .space  4*16
sint21  .space  4*16
sint22  .space  4*16
sint23  .space  4*16
sint24  .space  4*16
sint25  .space  4*16
sint26  .space  4*16
sint27  .space  4*16
```

```
sint28    .space    4*16
sint29    .space    4*16
sint30    .space    4*16

int0:       B       _myISP          ; 外部中断 0 入口点
            NOP
            RETE
int1:    RETE                       ; 外部中断 1
            NOP
            NOP
            NOP
int2:    RETE                       ; 外部中断 2
            NOP
            NOP
            NOP
tint0:    RETE                      ; 定时器 0 中断
            NOP
            NOP
            NOP
brint0:   RETE                      ; 多通道缓冲串口 0 接收数据中断
            NOP
            NOP
            NOP
bxint0:   RETE                      ; 多通道缓冲串口 0 发送数据中断
            NOP
            NOP
            NOP
brint1:   RETE                      ; 多通道缓冲串口 1 接收数据中断
            NOP
            NOP
            NOP
bxint1:   RETE                      ; 多通道缓冲串口 1 发送数据中断
            NOP
            NOP
            NOP
bint3:    RETE                      ; 外部中断 3
            NOP
            NOP
```

```
        NOP
        .end
```

可以在上电复位后，重新定位中断向量表的位置。中断向量的访问是严格按地址排列的。上面的程序中，每一个语句都不能少，也不能多，少一个 NOP 或多一个 NOP 都是不行的。结合表 2-17 可以看出，每个中断向量之间只有 4 个字的间隔。在 vectors.asm 中，中断向量的名字是随意命名的，但是相对位置是不可改变的。总之，中断向量表具有非常固定的格式，对于不同系列的 DSP 芯片，它们有一些不同，可以参考 VC5402 的中断向量表编写出相应的中断向量表程序。当然，在程序中，没有用到很多中断信号时，可以从后向前依次省略直到遇到用到的中断向量。上面程序中，仅给出了响应中断 INT0 的格式，其他中断响应的格式相同，保持中断间的间距为 4 个字。

2.5 本 章 小 结

通过本章的学习，至少应有以下几方面的心得：

(1) 应清楚面向 DSP 的程序设计的硬件基础是什么。因为 DSP 程序设计是面向实时处理的硬件编程，所以，弄清所要开发的 DSP 的各种资源是至关重要的，这些资源包括 CPU、存储器、中断向量表、各种寄存器设置（初始化方法还要在第三章中介绍）、片上外设等等，这些资源必须非常清楚才有可能设计出好的程序来。

(2) 应掌握怎么访问物理存储器、如何对映射存储器进行配置、如何编写 .cmd 文件和中断向量表的方法和技巧，它们是进行 DSP 程序设计必须具备的知识。例如，要想访问 DSP 中片上物理存储器，必须将这些物理存储器映射到映射存储器上才能访问，对于 DSP 内部寄存器的访问也是一样，这些方面与单片机是有区别的。又如，对于同一个系列的 DSP 芯片来说，.cmd 文件和中断向量表文件的格式是固定的，如果读者在使用 VC5402 芯片，那么本书上的这两个文件可以直接引用了。对于其他系列的芯片，需要根据实际情况做一点改动，但是原则和方法是一样的。如果用汇编语言编写程序，可使用 Visual Linker 进行可视化地编写存储器配置文件，扩展名为 .rcp。这对于 C/C++也是可以的。但因 C/C++语言相对于汇编语言来说具有较少的分配段，所以可以直接使用连接命令语言来编写 .cmd 文件，而且也很直观。

(3) 本章对三意电子公司的 SY-5402EVM 板作了一些介绍，特别是给出了语音处理部分的原理图，借此强调借助 EVM 板进行完整的程序设计的必要性。对于程序设计而言，一个完整的程序设计是从借助 EVM 板进行数据采集、算法评估和测试及数据发送开始的，一直进行到将合法的程序下载到 EVM 板上的 FLASH 存储器中，使 EVM 板脱离硬仿真器（Emulator）的环境独立地运行，或独立地运行并通过 Emulator 与计算机进行通信。

最后指出，对本章开始提出的第二、三个思考题前面已有了阐述，这里仅给出 TMS320VC5410 的存储器映射配置。TMS320VC5410 存储器映射配置是一个典型代表，它分成了三个独立的映射存储空间：64 K×16 bit 程序存储器、64 K×16 bit 数据存储器、64 K×16 bit IO 存储空间，且程序空间可以扩展为 128×64 K×16 bit。对这三个映射存储空间的

访问是通过存储器选择信号 PS、DS 和 IS 来指示的。IO 存储空间的寻址范围为 0000h 至 FFFFh，只存在于片外。程序存储空间和数据存储空间的映射图如图 2-18 所示。

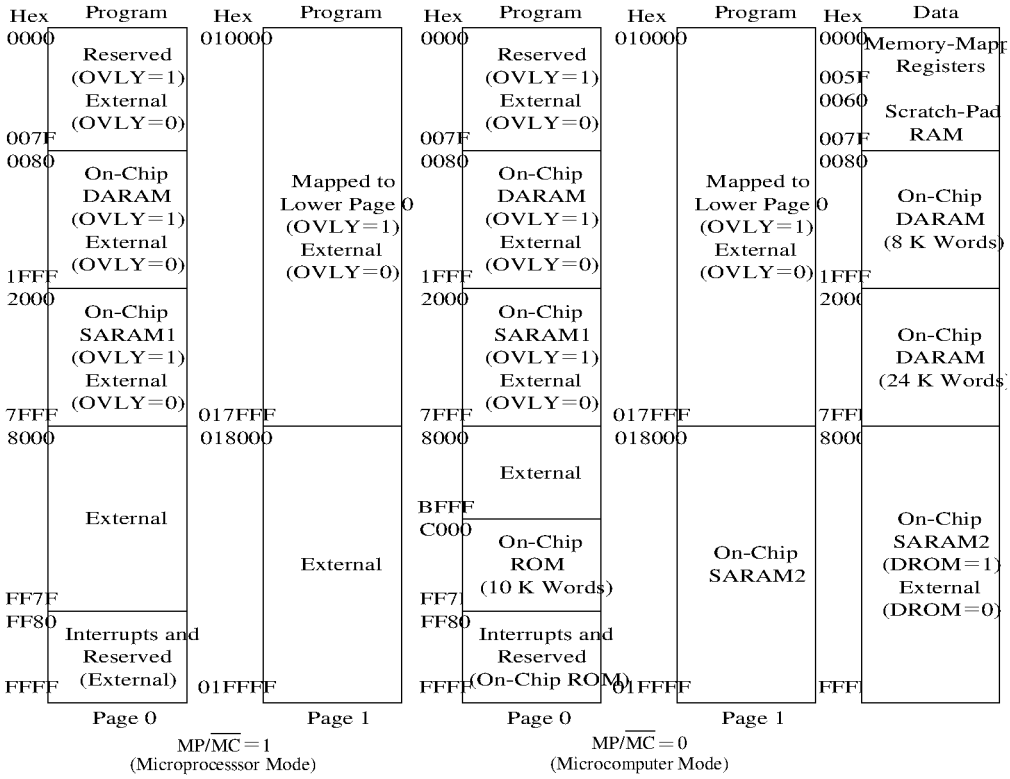


图 2-18　VC5410 的映射存储空间

习　题　二

1. 如何访问 DSP 片内的物理存储器？
2. 简述 TMS320VC5402 的 CPU 结构。
3. TMS320VC5402 的映射存储器结构是怎样的？存储器为什么要作映射？
4. 如何访问 DMA 控制寄存器？
5. 如何使用 DMA 进行数据传输？
6. 计时器中断是怎样产生的？
7. 如何初始化 McBSP 口？
8. DSP 如何访问外部低速数字化器件？
9. 并行 I/O 口如何访问 IO 空间？
10. 如何通过增强的主机接口进行数据传输？

11．简述 VC5402 的中断向量及其意义。
12．DSP 如何对串行口的数据进行访问？
13．IMR 和 IFR 有什么区别？各有什么用途？
14．简述 TMS320VC5510 的映射存储器分配，并指出它与 VC5402 的区别。
15．通过 EHPI16 进行数据传输的方法是什么样的？
16．SY-5402EVM 板上的两片 RAM 是做什么用的？为什么？
17．简述如何编写存储器配置文件。
18．根据本章中给出的 MAP 表（user_learn.map）说明表中各个部分的含义。
19．如何编写中断向量表文件？
20．如何在 MAP 表中查找 C 语言程序的入口地址？
21．TMS320VC5410 的映射存储器是怎样构成的？

第三章

C/C++程序设计

3.1 本章内容简介

本章介绍C语言的数据结构、库函数及优化的DSPLIB库函数；详细地介绍了实际C/C++语言开发C5000应用系统的编程方法、实例及程序框架，读者可以在此基础上修改相应程序并应用于其他的项目开发中；还进一步介绍了调试程序的各种技巧和方法。虽然CCStudio完全支持标准的C/C++语言，但考虑到本书的完整性，在本章仍然用了大量篇幅结合实际的DSP编程需要介绍了标准C/C++的数据结构和库函数。本章介绍的这些数据结构和库函数都是在SY-5402EVM板上测试通过的，对C6000也具有适用性和参考价值。

本章内容是本书的重点内容，围绕着使用标准C/C++语言编写C5000应用程序的方法这一主题展开论述。掌握了这一章的内容可以使读者熟练地运用C/C++语言进行DSP程序设计。本书前三章的内容是一个整体，自成体系，掌握了前三章的内容，应该基本具有了使用C/C++语言开发一个C5000系统平台的能力。

在此特别强调：面向DSP的C/C++程序设计中，数据结构是相对简单的，没有像通用计算机上的C/C++那样复杂的数据结构；算法（或函数或方法或功能）是非常重要的，在面向DSP的程序设计中，应力求实时性。因此，在DSP的结构设计中，数据区的空间相对于程序区一般要小得多。

思考题

(1) DSPLIB库函数的数据结构是什么类型？

(2) C/C++语言是如何访问VC5402片上外设的？

(3) 如何进行混合语言编程？

(4) 如何优化C/C++语言程序？

(5) C5000定点DSP支持浮点C/C++编程吗？

3.2 C/C++ 程序设计

3.2.1 面向 DSP 的 C/C++程序设计原则

1) 面向 DSP 的 C/C++程序设计与通用计算机上的 C/C++程序设计的比较

面向 DSP 的 C/C++程序设计与通用计算机上的 C/C++程序设计有很多不同之处，这也正是面向 DSP 的 C/C++程序设计的特色所在。在通用计算机上开发 C/C++语言程序，程序运行界面受到了高度的重视，目前已经出现了专门设计人机界面的程序开发人员。在 DSP 上编写 C/C++程序，是没有任何界面可言的，这时的人机接口是来自受 DSP 控制的终端，C/C++程序起到管理和控制的作用，类似于操作系统软件的作用；但是，面向 DSP 的 C/C++程序应属于应用程序的范畴。

通用计算机上的 C/C++语言程序与面向 DSP 的 C/C++语言程序最本质的区别在于：前者是大量数据的集中式处理过程，而后者是针对极少数据点的实时处理过程。计算机是将全部数据作为一个输入向量，进行足够长时间的处理，得出所需要的高精度的结果。在这个过程中，尽可能采用快速算法以节约时间，但是并不要求计算机仿真的时间与现实的时间相等，即不要求具有实时性。所谓实时性，主要是针对离散系统来讲的，即要求在采样时间间隔内，DSP 完成所有需要处理的数据处理任务，并处于空闲状态（或进程），等待下一个采样点的数据到来。数据到达后，根据信号处理算法的需要，可能会与前面到达的数据联合处理，也可能单独处理。不论采用哪一种处理方式，下一点数据到达之前的瞬间，该点数据所属的所有处理进程必须处理完毕，下一点的数据一旦到达，DSP 将开始下一点的数据处理。

另一种情况例外，就是并行处理。根据并行处理方式的不同，着眼点不同，实时性的含义略有差异。但就并行处理的一般含义而言，通常并不一定要求逐个数据点连续的进程的实时性，但要求多个并行进程必须是实时的。也就是说，面向 DSP 的 C/C++程序设计不像通用计算机那样单纯对数据流进行处理，它兼顾了数据流和时序机制的处理。时序机制是定义 DSP 工作能力的一个重要指标，包括了 DSP 的内部工作频率和 DSP 与所有外设进行通信的时钟频率，以及在时序驱动下的数据流格式定义等等。时序机制决定了 DSP 的实时处理能力，目前的一些 DSP 器件的时序机制能完成基带内的几乎全部数字信号处理。

通用计算机上的 C/C++程序设计有直观的输入和输出设备，可以直接观察运行的结果，无需借助一些示波器等等的仪器。而面向 DSP 的 C/C++程序设计是没有直观的输入、输出设备的，它的输入和输出均为映射存储空间的某个或某些地址及其这个地址中的数据。实际上，DSP 也只能访问（包括读和写）它的映射存储空间，虽然这个空间不一定是实在的东西，对这个空间的访问可以在 DSP 的外设上反映出来，这个反映必须借助于如数字示波器、逻辑分析仪等观测设备进行辅助分析。

通用计算机的 C/C++程序设计的数据来源可以由计算机的信号处理软件仿真产生，也可以是通过计算机接口接收外部的实时数据。如果时序机制允许的话，计算机也会实现一些实时运算，因此计算机可以对数据流进行集中处理，也可以完成一些低速实时处理。但

是面向 DSP 的 C/C++程序设计的数据来源只能是外部 A/D 送来的。DSP 的数据存储区是相当有限的，它不可能完成大量数据流的集中处理，即使是运算的中间结果，也不可能太多。

通用计算机上的 C/C++语言程序设计是要杜绝出现死循环的，而面向 DSP 的 C/C++程序设计却是必然出现死循环才行，这也是两者程序设计的又一个明显区别。

由于计算机的 CPU 和 DSP 的 CPU 在本质上和工作原理上是一致的，所以，面向 DSP 的 C/C++程序设计与通用计算机上的 C/C++程序设计又具有本质上的一致性，即有类似的编程风格、类似的程序框架、类似的编译执行过程，以及基本类似的设计思想。

2) 面向 DSP 的 C/C++程序设计原则

面向 DSP 的 C/C++程序设计，有一条基本的原则，即 C/C++程序不但需要对数据流进程进行编程，也要对时序机制进行编程，两者是同样重要的。在编程风格上，要求程序简练、高效。

就面向对象的 C++程序设计而言，其数据的封装作用等一些 C++的特性，对于通用计算机的程序设计是一种有效的方法。但是，在目前 DSP 速度不够高的情况下，C++的这些特性不一定有很大的优势，原因在于：其一，DSP 强调实时处理，容许的数据量小；其二，C++的编译执行效率可能没有 C 语言成熟。CCStudio 几乎支持标准 C++的所有语法，所有能在 Borland C++3.1 上调试通过的标准 C++程序（流库除外），可以不加修改地在 CCStudio 上运行。本章也给出了一个 C++的实例，建议在进行类的封装时将时序机制一起进行封装。

特别需要指出的是 CCStudio 不支持 C++的流库。CCStudio 对一些 C++库函数的支持，都仅仅是对 C 语言函数的少量扩充。不提倡完全采用 C++的方法来设计程序。如果可以，又有必要使用，当然应该用 C++ 来设计；如果没有必要用，就不一定非要用 C++来设计。但是，使用 DSP/BIOS 进行程序设计时，具有明显的 C++特性。对于 C++的程序，应使用 .cpp 的扩展名，也可在编译选项中进行设置。对于 C 程序，采用 .c 的扩展名。凡是 C 程序均可采用 C++编译器编译通过。

在 CCStudio 中，C 语言依然是 C++的一个真子集。因此在后文中，一般采用 C 程序来统指 C/C++程序。

3.2.2　C/C++程序设计流程

本章以采用 SY-5420EVM 板进行 C 程序设计为平台，详细介绍 C 程序的设计流程。采用其他 DSP 功能板的 C 程序设计流程与此流程是完全类似的。C 程序设计流程一般分为以下几步：

? 第一步，设计平台准备。软件工程师要根据开发项目的实际需要选用相应的 DSP 功能板，并要对 DSP 功能板中需完成的算法的相关硬件有全面的了解，对于 DSP 与这些硬件的接口更应有明确的数据流和时序机制定义。

面向 DSP 的 C 程序设计是基于硬件的设计，忽视了对硬件的理解、单纯强调软件的实现是不合理的。例如，要实现一个语音信号的实时算法，选用了三意电子的 SY-5402EVM 板，与实现这个算法相关的硬件有 TMS320VC5402、EPM7128SLC84-15（是一片 15 ns 的 CPLD，在 SY-5402EVM 板上，A/D/A 的数据和时序都经过该 CPLD 作缓冲）、TLC320AD50C

（A/D/A）、TLC274（功放）等。设计者的第一步工作就是阅读这些器件的芯片资料，把数据流和时序机制关系弄清楚。

？第二步，数据流和时序机制的编程。在第一步工作的基础上，编写整个系统的初始化 C 程序以及对 DSP 端口进行数据访问的 C 程序。

这一步的编程，目的是实现整个系统的初始化工作，并保证在时序机制控制下数据通道的完整性，即为算法处理准备了输入和输出数据。例如，对于 SY-5402EVM 板来说，初始化工作应包括烧写 CPLD、初始化 VC5402 和 AD50 等（其中，SY-5402EVM 板的 CPLD 在出厂时已烧写好了时序逻辑，且是近似最优的。请参考 evm5402.gdf 文件，用 MAXPLUSII 打开。其与语音处理相关的逻辑很少，而且简单。当然也可以重新配置 CPLD）。与 AD50 进行通信的是 VC5402 的 McBSP1，初始化 VC5402 时主要应完成 CPU 和 McBSP1 的初始化。AD50 可以采用上电硬复位后的配置，也可以使用 VC5402 进行重新配置。

？第三步，数据处理算法的编程。数据处理算法在上机测试之前，应有一个明确的流程图。如果算法完全是一种新的方法，应先用 MATLAB 等软件作算法的仿真试验。但并非完全不可以直接使用 DSP 功能板进行试验，这主要是取决于软件工程师具备的开发环境优劣、算法设计的正确性和对硬件的编程控制能力。

对于实时处理算法的设计应符合一条从简单到复杂的基本原则：首先，设计简单的算法来验证输入、输出数据的正确性；其次，采用通用的 C 语言编写全部的算法，在编写过程中不断调试，直到完成整个算法；然后，尽量采用 CCStudio 提供的 DSPLIB 库函数进行算法的优化，或者在前一步中直接使用这些库函数；最后，编译优化并完成整个算法。特别需要指出的是，在整个过程中，应对所有的程序和算法进展情况有详细的文档和清单，有助于连续思维，这样常常有事半功倍的效果，也便于与其他设计人员讨论或与其他人员合作。这一步的算法设计与通用 C 语言编程的算法设计方法具有相似性。

？第四步，通信试验。以上工作是在仿真器环境下实现的，这一步的工作是将程序烧写到 DSP 功能板上，进行实际的通信试验。

下面给出了面向 DSP 的 C 程序设计流程图，如图 3-1 所示。

对图 3-1 需要说明的一点是：当 Boot Loader 之前的部分流程工作正确时，一般来说 Boot Loader 之后的部分流程能正确工作；但是当通信试验不成功时，可能要从头检查程序的正确性，这时就会发现文档工作的重要性。

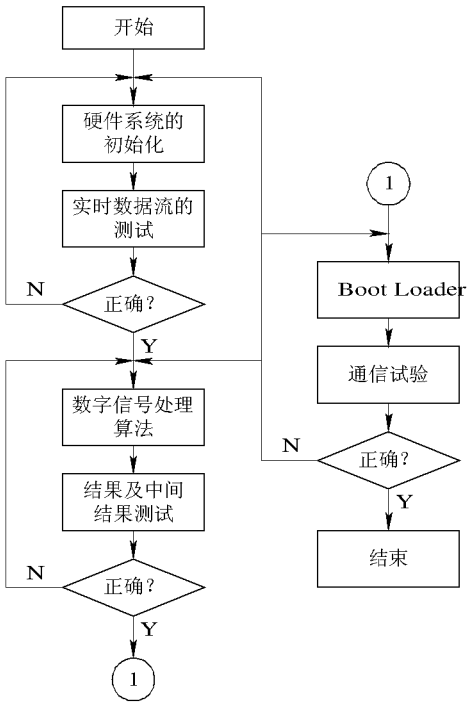


图 3-1 面向 DSP 的 C 程序设计流程图

3.2.3　C/C++程序设计框架

C 语言程序具有固定的格式。C 语言是一种函数式的语言，一个完整的 C 程序是由一个主函数（main()函数）和一些完成特定算法的子函数组成。C 程序的入口点固定为 c_int00，由 main()函数指定，由 rts.lib 库定义，其他子函数功能的实现必须经过 main()函数的调用才行。此外，C 语言支持丰富的数据结构类型和各种扩展的数据结构。因此，C 程序=C 语言数据结构+C 语言函数（+C 时序控制），其中，C 语言函数可以采用 C 或 DSP 的库函数，也可以自己编写子函数。使用 C 或 DSP 的库函数时，必须包括相应的运行库。C 语言函数包括用于时序机制处理的语句和用于数据处理的语句。

下面用伪代码的形式给出 C 程序的通用框架：

```
//主程序文件的内容，扩展名为.c 或.cpp
#include  “函数库 1”
#include  <函数库 2>
#include  “自定义函数库 3”
…
#include  “自定义函数库 n”
//上面将所有需要调用的函数列于文件头部
//下面定义所有使用的宏名称
#define  宏名 1  指代内容
…
#define  宏名 n  指代内容
//下面为全局变量，变量类型中包括自定义变量类型
变量类型 1    全局变量名 1;
变量类型 2    全局变量名 2;
…
变量类型 n    全局变量名 n;
//下面为本程序内部的函数声明，这些函数的例程一般放在 main()函数后面
函数类型 1  函数名 1(函数变量类型列表 1，不用列出函数变量名表);
//例如：DATA  func_learn(unsigned short * , int , short);
函数类型 2  函数名 2(函数变量类型列表 2);
…
函数类型 n  函数名 n(函数变量类型列表 n);
//下面为主函数
函数返回值类型  main(参数类型列表，主函数一般不设)
```

```
{
    //局部变量定义，作用域仅为从此开始到本函数结束
    变量类型 1    局部变量名 1;
    变量类型 2    局部变量名 2;
    …
    变量类型 n    局部变量名 n;
    //下面为一些算术运算
    …
    //可以调用子函数来处理数据
    …
    //完成数据的输入输出
    …
    //在 C/C++中，对变量和子函数都是先定义后使用的原则。在 C 中更为严格一些，
    //变量定义必须放在程序头部!! 其他的语句视具体需要安排先后次序
}
//下面为本程序中定义的子函数，这些子函数前面已经声明了
函数类型 1   函数名 1(函数变量类型和变量名列表 1)
{
    //本函数用到的局部变量
    …
    //本函数中的算法
    …
}
函数类型 2   函数名 2(函数变量类型和变量名列表 2)
{
    //本函数用到的局部变量
    …
    //本函数中的算法
    …
}
…
函数类型 n   函数名 n(函数变量类型和变量名列表 n)
{
    //本函数用到的局部变量
    …
```

```
    //本函数中的算法
    …
}
//主程序文件结束

//一个子程序文件的框架，文件扩展名为.h 或.hpp，本文件应被主文件包括，
//即将 C 语句　#include　“本子程序文件名”加入到主程序文件中
//下面为应用于本文件中的变量定义
…
//下面为一些用到的宏定义
…
//下面为引用其他文件的包括语句
#include　“函数库文件 1”
#include　<函数库文件 2>
#include　“自定义函数库 3”
…
#include　“函数库文件 n”
//下面为实现特定功能的子函数
…
//本子程序结束，其他的子程序定义与此类似。上述定义中有些用不到时可以省略
```

下一节中将给出一个完整的 C 程序示例，以帮助读者加深对这部分的理解。

3.3　C 程序设计示例

3.3.1　硬件准备及实现结果

1) 硬件准备

本程序示例的硬件平台包括计算机、DSP 仿真器、三意电子 SY-5402EVM 板、耳机、计算机声卡输出及 SY-5402EVM 板声频输入端的音频连接线以及电源等。特别强调的是，使用普通的麦克风是不行的！硬件上需要做一点调整，即将 SY-5402EVM 板上的 CLKMD1、CLKMD2、CLKMD3 设置为 110（原来的设置为 001）。因为 SY-5402EVM 板上使用的是 10 MHz 的外部时钟源，原来的设置是上电复位时就使 VC5402 工作在 100MHz 时钟下（PLL×10 模式）。作了这些调整设置后，上电复位工作在 PLL×1 模式下，即上电后 VC5402 工作在 10 MHz 时钟下，再通过初始化程序将时钟频率倍频到 100 MHz。这样改变的原因是考虑到采用改变后的设置 VC5402 工作会更加稳定。

SY-5402EVM 板上有很多跳线，这些跳线增加了设计开发的灵活性。与本程序相关的三个跳线是 JP11、JP1 和 JP6。其中，JP11 控制输入语音信号的模式，设置为交流方式；JP1 控制 CPLD（EPM7128）是否使 AD50 与 VC5402 相连通，JP1 设为高电平；JP6 为 VC5402 的 MP/MC 脚，出厂时设为 MP，读者可将其设为 MC，即使得 MP/MC=0。AD50 的串行时钟和帧时钟均为输出信号，若采用了上电时 AD50 的初始配置，则初始化 VC5402 的串行口 1 的所有时钟信号为输入模式。整个系统工作在串行时钟为 2.5 MHz、采样率为 9.765625 kHz 的时序机制下（这是推算得来的，没有实测）。AD50 采用 15 位数据的传输格式，因此 VC5402 收到 AD50 的数据后，将最后一位（指最低位）置 0；而 AD50 要求 VC5402 发送数据时，必须将最后一位（指最低位）置 0 后再发送（这些是由 AD50 数据资料上得来的，限于篇幅，请读者参考 AD50 的器件资料）。

2) 实现结果

本示例程序最终完成的结果是：通过计算机声卡将语音信号（模拟）送到 SY-5402EVM 板语音信号输入端，采用音频播放器播放一首歌曲，将耳机接到 SY-5402EVM 板的语音信号输出端，此时可以通过耳机听到十分清晰的歌声。同时，在设计的程序中还访问了通用 XF 脚，使 XF 控制 LED（SY-5402EVM 板上的 D3）灯闪烁。

3.3.2 程序分析

若将本示例程序与三意电子公司提供的类似功能的程序（或是 CCStudio 提供的原程序例程）作对比，从中可以看出本示例程序具有广泛的通用性和代表性，可以应用于 C5000 或是 C6000 系列上去，这在编程方法上也具有指导意义。

程序的对比分析如表 3-1 所示。

表 3-1 本章示例程序与三意电子的原程序的对比

对比项目	本章示例程序	三意电子的程序
库函数调用	仅调用 C 语言通用支持库 rts.lib	除了调用了 rts.lib 外，还调用了专用 VC5402EVM 的库 drv5402.lib 和 dsk5402.lib
包括函数	自定义的函数，具有源程序代码	包括了一些封装好了的专用于 VC5402EVM 的库函数，仅有调用方法，没有具体的源代码
头文件.h	定义了自己的头文件，并使用了通用 C 语言的头文件	使用了 VC5402EVM 提供的专用头文件
通用性	具有通用性	通用性差
面向 DSP 的编程思想	具有指导意义	指导意义差

这种比较并不意味着本章示例程序执行效果比三意电子的示例程序执行效果好（作者使用”裁缝师”的测试结果表明两个程序性能差不多），只能说明本章示例程序具有面向 DSP 的 C 程序设计的通用性和代表性。

本章示例程序的工程文件如图 3-2 所示，包括 C 主程序 user_audio.c、汇编语言中断向量表文件 vectors.asm、C 语言运行支撑库 rts.lib（CCStudio 系统库）、存储器配置文件 user_audio.cmd 以及三个头文件 user_type.h、user_face.h、user_func.h。其中，user_type.h 定义所需要的自定义变量类型；user_face.h 定义 VC5402 的接口，对于其他型号的 DSP，这部分要作适当的调整；user_func.h 中定义了几个用户自定义函数；user_audio.c 是主程序文件，完成对 LED 灯闪烁的控制及输入/输出数据的简单传送；vectors.asm 和 user_audio.cmd 文件前面已详细介绍。为了完整性，下一节将再给出源程序，供读者参考。

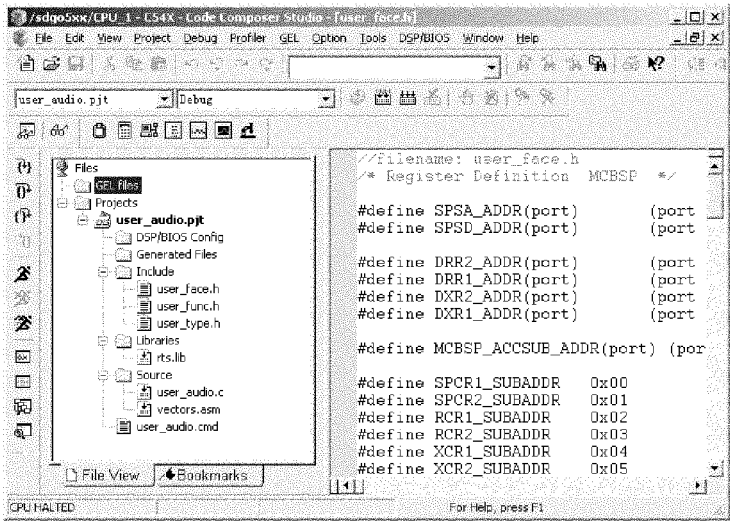


图 3-2　本章示例程序的工程文件

运行示例程序暂停后的寄存器的值如图 3-3 所示。在 Watch 窗口中显示的是 CLKMD 的值，使用 Watch 窗口可以观测任何地址。例如，对于 CLKMD 寄存器，它映射在数据存储区的 0x0058 处，只要在 Watch 窗口中输入*(0x0058)即可。需要注意的寄存器有 MP/MC、PMST、DRR1、DXR1、OVLY、ST1、XF 等，具体含义请参考在线“帮助”。

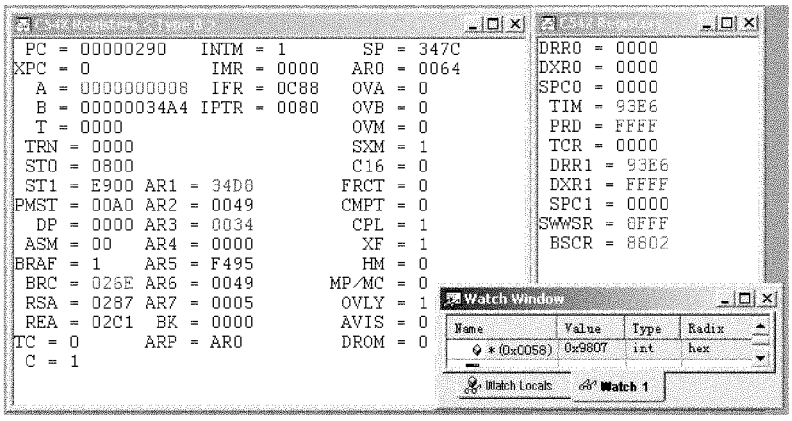


图 3-3　示例程序运行时的一些寄存器值

3.3.3　程序源代码

下面给出了全部的源程序内容，其中文部分是作者为了进一步解释程序代码而加入的注解，原程序中不存在这些中文注释。

1．头文件

三个头文件依次为 user_type.h、user_face.h 和 user_func.h，下面列出了其源程序代码，以供参考。

下面为 user_face.h。

```
//filename: user_face.h
/* Register Definition    MCBSP    */

#define SPSA_ADDR(port)          (port ? 0x48 : 0x38)
#define SPSD_ADDR(port)          (port ? 0x49 : 0x39)

#define DRR2_ADDR(port)          (port ? 0x40 : 0x20)
#define DRR1_ADDR(port)          (port ? 0x41 : 0x21)
#define DXR2_ADDR(port)          (port ? 0x42 : 0x22)
#define DXR1_ADDR(port)          (port ? 0x43 : 0x23)

#define MCBSP_ACCSUB_ADDR(port) (port ? 0x49 : 0x39)

#define    SPCR1_SUBADDR          0x00
#define    SPCR2_SUBADDR          0x01
#define    RCR1_SUBADDR           0x02
#define    RCR2_SUBADDR           0x03
#define    XCR1_SUBADDR           0x04
#define    XCR2_SUBADDR           0x05
#define    SRGR1_SUBADDR          0x06
#define    SRGR2_SUBADDR          0x07
#define    MCR1_SUBADDR           0x08
#define    MCR2_SUBADDR           0x09
#define    RCERA_SUBADDR          0x0A
#define    RCERB_SUBADDR          0x0B
#define    XCERA_SUBADDR          0x0C
#define    XCERB_SUBADDR          0x0D
#define    PCR_SUBADDR            0x0E
//上面定义了 McBSP 各端口的地址及子地址，下面为这些端口的英文解释
/*
```

VC5402----------McBSP0　McBSP1

Name	Address	Name	Address	Sub-Address	Description
SPCR10	39h	SPCR11	49h	00h	Serial port control register 1
SPCR20	39h	SPCR21	49h	01h	Serial port control register 2
RCR10	39h	RCR11	49h	02h	Receive control register 1
RCR20	39h	RCR21	49h	03h	Receive control register 2
XCR10	39h	XCR11	49h	04h	Transmit control register 1
XCR20	39h	XCR21	49h	05h	Transmit control register 2
SRGR10	39h	SRGR11	49h	06h	Sample rate generator register 1
SRGR20	39h	SRGR21	49h	07h	Sample rate generator register 2
MCR10	39h	MCR11	49h	08h	Multichannel register 1
MCR20	39h	MCR21	49h	09h	Multichannel register 2
RCERA0	39h	RCERA1	49h	0Ah	Receive channel enable register partition A
RCERB0	39h	RCERB1	49h	0Bh	Receive channel enable register partition B
XCERA0	39h	XCERA1	49h	0Ch	Transmit channel enable register partition A
XCERB0	39h	XCERB1	49h	0Dh	Transmit channel enable register partition B
PCR0	39h	PCR1	49h	0Eh	Pin control register

```
*/
//下面定义各个端口的初始化值
//SPCR10   39h SPCR11   49h  00h  Serial port control register 1
#define    bsp_SPCR11        0x0021
//SPCR20   39h SPCR21   49h  01h  Serial port control register 2
#define    bsp_SPCR21        0x0201
//PCR0     39h PCR1     49h  0Eh  Pin control register
#define    bsp_PCR1          0x000C
//RCR10    39h RCR11    49h  02h  Receive control register 1
#define    bsp_RCR11         0x0040
//RCR20    39h RCR21    49h  03h  Receive control register 2
#define    bsp_RCR21         0x0000
//XCR10    39h XCR11    49h  04h  Transmit control register 1
#define    bsp_XCR11         0x0040
//XCR20    39h XCR21    49h  05h  Transmit control register 2
#define    bsp_XCR21         0x0000
//SRGR10   39h SRGR11   49h  06h  Sample rate generator register 1
#define    bsp_SRGR11        0x0000
//SRGR20   39h SRGR21   49h  07h  Sample rate generator register 2
#define    bsp_SRGR21        0x0000
//MCR10    39h MCR11    49h  08h  Multichannel register 1
#define    bsp_MCR11         0x0000
```

```
//MCR20        39h  MCR21   49h  09h  Multichannel register 2
#define        bsp_MCR21          0x0000
//RCERA0       39h  RCERA1  49h  0Ah  Receive channel enable register partition A
//#define      RCERA1
//RCERB0       39h  RCERB1  49h  0Bh  Receive channel enable register partition B
//#define      RCERB1
//XCERA0       39h  XCERA1  49h  0Ch  Transmit channel enable register partition A
//#define      XCERA1
//XCERB0       39h  XCERB1  49h  0Dh  Transmit channel enable register partition B
//#define      XCERB1
/* McBSP 口数据收发寄存器的地址及意义
Name    Address    Type          Description
DRR20   20h        McBSP #0      McBSP0 data receive register 2
DRR10   21h        McBSP #0      McBSP0 data receive register 1
DXR20   22h        McBSP #0      McBSP0 data transmit register 2
DXR10   23h        McBSP #0      McBSP0 data transmit register 1
DRR21   40h        McBSP #1      McBSP1 data receive register 2
DRR11   41h        McBSP #1      McBSP1 data receive register 1
DXR21   42h        McBSP #1      McBSP1 data transmit register 2
DXR11   43h        McBSP #1      McBSP1 data transmit register 1
*/
/*
SPSA0   38h        McBSP #0      McBSP0 subbank address register
SPSD0   39h        McBSP #0      McBSP0 subbank data register
SPSA1   48h        McBSP #1      McBSP1 subbank address   register
SPSD1   49h        McBSP #1      McBSP1 subbank data register
*/
//---------CPU----------
//ST1     addr:0x0007    Status register 1   ST1[13]=XF
#define   reg_ST1        0x0007
#define   ST0            *(volatile unsigned int*)0x06
#define   ST0_ADDR           0x06

#define   PMST           0x001D
#define   SWWSR          0x0028
#define   SWCR           0x002B
#define   BSCR           0x0029
#define   CLKMD          0x0058
```

```
#define   PMST_VAL      0x00A0   //interupt vectors from ox80
#define   SWWSR_VAL     0x8fff
#define   SWCR_VAL      0x0001
#define   BSCR_VAL      0x8802
#define   CLKMD_VAL     0x9FF7
```

下面为 user_type.h，它自定义了一些数据类型，同时展示了自定义数据类型的方法，有些示例类型在本章程序中没有使用。

```
//filename: user_type.h
typedef float f32;
typedef long s32;
typedef int s16;
typedef unsigned char u8;
typedef unsigned int u16;
typedef unsigned long u32;
```

下面为 user_func.h。

```
//filename: user_func.h
/*void delay(s16 period)     这个函数定义延时功能，在本程序中未用上
{
      int i, j;
      for(i=0; i<period; i++)
      {
            for(j=0; j<period; j++);
      }
}*/

void init_board(void)
{
      *(volatile u16 *)CLKMD = 0x0000;
      while(*(volatile u16 *)CLKMD & 0x0001){};
      *(volatile u16 *)CLKMD = CLKMD_VAL;   //初始化 VC5402 时钟为 100 MHz

      *(volatile u16 *)PMST = PMST_VAL;
      *(volatile u16 *)SWWSR =SWWSR_VAL;
      *(volatile u16 *)SWCR =SWCR_VAL;
      *(volatile u16 *)BSCR =BSCR_VAL;        //初始化 CPU 寄存器
```

```
    *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //下面为初始化串行口 1
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SPCR11;
    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SPCR21;
    *(volatile u16 *)SPSA_ADDR(1)=RCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_RCR11;
    *(volatile u16 *)SPSA_ADDR(1)=RCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_RCR21;
    *(volatile u16 *)SPSA_ADDR(1)=XCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_XCR11;
    *(volatile u16 *)SPSA_ADDR(1)=XCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_XCR21;
    *(volatile u16 *)SPSA_ADDR(1)=SRGR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SRGR11;
    *(volatile u16 *)SPSA_ADDR(1)=SRGR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SRGR21;
    *(volatile u16 *)SPSA_ADDR(1)=MCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_MCR11;
    *(volatile u16 *)SPSA_ADDR(1)=MCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_MCR21;
    *(volatile u16 *)SPSA_ADDR(1)=PCR_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_PCR1;
}

// volatile ioport unsigned port400;    下面的程序仅供参考，是作为 LED 测试用的
/*void LEDTest(unsigned int count)
{
    int a;
    while(count)
    {
        a=0x5;
        port400=a;
        delay(80);
        a>>=1;
        port400=a;
        delay(80);
        count--;
    }
    port400=0x0;
}*/
```

2．主程序文件

下面列出主程序文件 user_audio.c 的源代码供参考。

```
/*********************************************************************/
/* filename: user_aduio.c                                            */
/* Author:ZhangYong                                                  */
/* 2002-12                                                           */
/*********************************************************************/
#include "user_type.h"
 #include "user_face.h"
 #include "user_func.h"            //包括的头文件

 /* Global Variables               全局变量    */
 s16 in_data;                      /*temp data*/
 s16 out_data;
 s16 aud_data[100];
 // Main program
 void main()                       //主函数入口地址
 {
       unsigned int i=0;
       unsigned int j=0;
       init_board();               //调用初始化函数对整个 SY-5402EVM 板初始化

       while (1)
 {

            //Receive Data from McBSP1   从串口 1 收端接收数据
            *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR;
            while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
            in_data=*(volatile u16*)DRR1_ADDR(1);
            in_data &= 0xFFFE;         //这一句是适应 A/D/A 的要求
            aud_data[j]=in_data;       //将 100 个数据存入这个数组中进行内部测试
            j++;
            if(j>=100) j=0;
                  //add fir filter in the next part   后面的程序在此加入一些代码
                  //                                  完成其他的功能如 AGC 等
            out_data=in_data & 0xFFFE;      //这一句是适应 A/D/A 的要求
```

```
            //Transmit Data To McBSP1    下面是通过串口 1 发端发送数据
            *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;
            while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
            *(volatile u16 *)DXR1_ADDR(1)=out_data;
            //下面为每 1024 次 LED 灯闪烁一下
            i++;
            if(i>1024)
            {
                //      *(int *)(0x0007)^=0x2000;
                *(volatile u16 *)reg_ST1 ^= 0x2000;     //Every 1024 times XF changed
                i=0;
            }
        }
}
```

3．中断向量文件

下面列出了中断向量文件 vectors.asm。

```
            .sect ".vectors"

            .ref _c_int00                   //定义在 rts.lib 库中

            .align   0x80                   //中断向量表必须分配在 128 个字的整倍数区域段内

RESET:                                      //DSP 硬复位的中断，优先级最高
            BD _c_int00
            STM #128,SP
nmi:        RETE                            //下面依次为 VC5402 的各个中断，每个只占 4 个字
              NOP
              NOP
              NOP
sint17      .space      4*16
sint18      .space      4*16
sint19      .space      4*16
sint20      .space      4*16
sint21      .space      4*16
sint22      .space      4*16
sint23      .space      4*16
sint24      .space      4*16
```

```
sint25      .space      4*16
sint26      .space      4*16
sint27      .space      4*16
sint28      .space      4*16
sint29      .space      4*16
sint30      .space      4*16

int0: RETE                          //INT0 中断
            NOP
            NOP
            NOP
int1: RETE                          //INT1 中断
            NOP
            NOP
            NOP
int2: RETE                          //INT2 中断
            NOP
            NOP
            NOP
tint0:RETE                          //计时器中断
            NOP
            NOP
            NOP
brint0:RETE                         //串口 0 接收数据中断
            NOP
            NOP
            NOP
bxint0:RETE                         //串口 0 发送数据中断
            NOP
            NOP
            NOP
brint1:RETE                         //串口 1 接收数据中断
            NOP
            NOP
            NOP
bxint1:RETE                         //串口 1 发送数据中断
            NOP
            NOP
            NOP
```

```
bint3:RETE                    //INT3 中断
        NOP
        NOP
        NOP
      .end
```

4．存储器配置文件

下面列出存储器配置文件 user_audio.cmd 的源代码。

```
-m   user_audio.map      /*编译连接时产生 MAP 表*/

MEMORY
{
    PAGE 0:   PROG(RWXI):  origin=0x180,   length=0x3000
              VECS(RXI):   origin=0x80,    length=0x100
/* VECS 存放中断向量表，长度至少为 0x80，当然 len=0x80 最佳*/
    PAGE 1:   DATA(RWI):   origin=0x3080,  length=0xF80
}
/*上面为存储区域的分配，下面为段的分配，在段中可以加入.data 区 */
SECTIONS
{
    .text      {} > PROG     PAGE 0
    .cinit     {} > PROG     PAGE 0
    .pinit     {} > PROG     PAGE 0
    .vectors   {} > VECS     PAGE 0
    .stack     {} > DATA     PAGE 1
    .bss       {} > DATA     PAGE 1
    .const     {} > DATA     PAGE 1
    .switch    {} > DATA     PAGE 1
    .system    {} > DATA     PAGE 1
    .cio       {} > DATA     PAGE 1
    .far       {} > DATA     PAGE 1
}
```

5．MAP 表文件

在本节的程序中，特别指出 PMST 寄存器的值决定了中断向量表重定位的位置。PMST（15：7）的值补上 7 个零为中断向量表在复位后的新位置，这个位置在存储器配置中要声明，在中断向量表中就加入 .align 0x80 保证其从一个 128 字的边界开始。

下面给出了生成的MAP文件，这可以使读者对实际的存储器占用情况有进一步的了解。

```
******************************************************************************
TMS320C54x COFF Linker                  PC Version 3.70
******************************************************************************
>> Linked Tue Dec 17 16:49:22 2002

OUTPUT FILE NAME:     <./Release/user_audio.out>
ENTRY POINT SYMBOL: "_c_int00"    address: 00000180


MEMORY CONFIGURATION

           name            origin          length          used            attr        fill
           ----------------  --------        --------        --------        ---------   -------
PAGE   0: VECS            00000080        00000100        00000064        RIX
           PROG            00000180        00003000        0000014f        RWIX

PAGE   1: DATA            00003080        00000f80        00000488        RWI
/*从上面可以看出程序入口地址为 0x180，本示例程序所占空间很小 */

/*下面是各个段在存储器内部的分配情况*/
SECTION ALLOCATION MAP

output                                                    attributes/
section     page    origin          length          input sections
--------    ----    ----------      ----------      ----------------
.text       0       00000180        00000148
                    00000180        00000045        rts.lib : boot.obj (.text)
                    000001c5        00000000        vectors.obj (.text)
                    000001c5        0000003d        rts.lib : exit.obj (.text)
                    00000202        000000c6        user_audio.obj (.text)

.cinit      0       000002c8        00000007
                    000002c8        00000006        rts.lib : exit.obj (.cinit)
                    000002ce        00000001        --HOLE-- [fill = 0000]

.pinit      0       00000180        00000000

.vectors    0       00000080        00000064
                    00000080        00000064        vectors.obj (.vectors)

.stack      1       00003080        00000400        UNINITIALIZED
                    00003080        00000000        rts.lib : boot.obj (.stack)
```

.bss	1	00003480	00000088	UNINITIALIZED
		00003480	00000000	rts.lib : boot.obj (.bss)
		00003480	00000000	vectors.obj (.bss)
		00003480	00000022	rts.lib : exit.obj (.bss)
		000034a2	00000066	user_audio.obj (.bss)
.const	1	00003080	00000000	UNINITIALIZED
.switch	1	00003080	00000000	UNINITIALIZED
.system	1	00003080	00000000	UNINITIALIZED
.cio	1	00003080	00000000	UNINITIALIZED
.far	1	00003080	00000000	UNINITIALIZED
.data	1	00000000	00000000	UNINITIALIZED
		00000000	00000000	rts.lib : boot.obj (.data)
		00000000	00000000	vectors.obj (.data)
		00000000	00000000	user_audio.obj (.data)
		00000000	00000000	rts.lib : exit.obj (.data)

/*下面为全局符号在存储器内的位置，这些信息可方便用户调试。其中给出了两种排列方式，*/
/*即按字符顺序和按地址顺序排列*/

GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

address	name
--------	----
00003480	.bss
00000000	.data
00000180	.text
000001ff	C$$EXIT
00000400	_STACK_SIZE
00003480	_bss_
000002c8	_cinit_
00000000	_data_
00000000	_edata_
00003508	_end_
000002c8	_etext_
ffffffff	_pinit_
00000180	_text_

```
000034a1    _cleanup_ptr
00000001    _lflags
000001ff    _abort
000001e8    _atexit
000034a4    _aud_data
00000180    _c_int00
000001c5    _exit
000034a2    _in_data
00000202    _init_board
0000027b    _main
000034a3    _out_data
000002c8    cinit
00000000    edata
00003508    end
000002c8    etext
ffffffff    pinit

/*下面是按符号的地址顺序排列的全局符号*/
GLOBAL SYMBOLS: SORTED BY Symbol Address

address     name
--------    ----
00000000    _data_
00000000    _edata_
00000000    edata
00000000    .data
00000001    _lflags
00000180    _text_
00000180    .text
00000180    _c_int00
000001c5    _exit
000001e8    _atexit
000001ff    C$$EXIT
000001ff    _abort
00000202    _init_board
0000027b    _main
000002c8    _cinit_
000002c8    _etext_
000002c8    cinit
000002c8    etext
00000400    _STACK_SIZE
00003480    .bss
```

```
00003480    _bss__
000034a1    _cleanup_ptr
000034a2    _in_data
000034a3    _out_data
000034a4    _aud_data
00003508    end
00003508    _end_
ffffffff    pinit
ffffffff    _pinit_

[29 symbols]
/*本示例中一共有 29 个符号*/
```

3.4　C/C++语言数据结构及语法

这一节按循序渐进的方法由浅入深地介绍面向 DSP 的 C/C++语言及其程序设计方法，以利于读者理解和掌握。

3.4.1　C/C++数据结构

1．C/C++数据类型

C54x 支持的基本数据类型分别如表 3-2 所示。C55x 除了支持表 3-2 中的所有数据类型外，还增加了表 3-3 所示的数据类型。

表 3-2　C54x 支持的基本数据类型

数据类型	字长(位)	表示意义	最小值	最大值
signed char	16	ASCII	-32768	32767
char, unsigned char	16	ASCII	0	65535
short, signed short	16	二进制补码	-32768	32767
unsigned short	16	二进制	0	65535
int, signed int	16	二进制补码	-32768	32767
unsigned int	16	二进制	0	65535
long, signed long	32	二进制补码	-2147483648	2147483647
unsigned long	32	二进制	0	4294967295
enum	16	二进制补码	-32768	32767
float	32	IEEE-32 bit	1.175494e-38	3.40282346e+38
double	32	IEEE-32 bit	1.175494e-38	3.40282346e+38
long double	32	IEEE-32 bit	1.175494e-38	3.40282346e+38
*(指针类型)	16	二进制	0x0000	0xFFFF

表 3-3　C55x 增加的基本数据类型

数据类型	字长(位)	表示意义	最小值	最大值
long long	40	二进制补码	-549755813888	549755813887
unsigned long long	40	二进制	0	1099511627775
小模式下的数据指针	16	二进制	0x0000	0xFFFF
大模式下的数据指针	23	二进制	0	0x7FFFFF
函数指针	24	二进制	0	0xFFFFFF

由表 3-2 和表 3-3 可以构造出 C5000 支持的自定义数据类型和扩展的数据类型。自定义数据类型的方法使用 typedef 类型说明符，扩展的数据类型包括：数组、结构体、共用体等。此外，C5000 系列还支持空类型。

2．C/C++常量与变量

认识了 C5000 支持的数据类型之后，需要进一步认识 C/C++定义常量及变量的方法。因为进行 C 语言程序设计参与运算的数据只有常量和变量两种类型，掌握了定义 C/C++的常量及变量的方法之后，就可编写简单的 C/C++程序了。

1) 定义常量

先介绍定义常量的方法，例如：

```
const  short  d1=8;  //C/C++程序中采用“//”注释一行，采用“/*  */”注释多行。
//上式为定义常量 d1，其值为 8，这种定义方法一定要初始化。
//下面的定义方法是不对的，即错误的
const  short  d1;
#define   SIN_0   0   //这是采用宏定义的方法定义符号常量，即程序中出现 SIN_0 时均会以 0 代替
```

程序中出现的数值也是常量，例如十进制数 100。另外，八进制数用“0”开头（不是字母 O），十六进制数以“0x”开头。字符常量用单引号括起，切记单引号中只能有一个字符，如'a'是正确的字符常量，'abc'是不合法的表示。在 C5000 中，'abc'= ='c'；字符串常量用双引号括起，如"Great Wall"等。C5000 中，一个整数常量是可以赋值给各种整型变量的，一个浮点数常量可以赋值给所有的浮点数类型。

2) 定义变量

下面罗列出变量的定义方法：

```
char  ch_1;            //定义一个字符变量，变量名为 ch_1
short  sh_temp;        //定义一个 short 型变量，变量名为 sh_temp
long  lg_dat1;         //定义一个常整型变量，变量名为 lg_dat1
float  fl_dat2;        //定义一个浮点型变量，变量名为 fl_dat2
short  * pt_addr1;     //定义一个指向 short 型数据的指针，指针名为 pt_addr1
```

常量名和变量名应当使用字母或是以下划线开头。最多可以有 100 个字符，不能使用 C 语言的关键字来作为常量名或变量名。对于变量名的指令，常采用著名的匈牙利命名规则。这个规则指出，对应于不同数据类型的变量名应给定相应的变量名头标志，并且符合见名

知义的原则。例如，用 f 表示浮点型数据，则定义浮点型变量 using 时，可以定义为 fUsing。读者没有必要一定采用匈牙利规则中指定的变量头标志，但是应尽可能地采用这种方法。

例如：

```
short   sh_a[10];          //定义了一个数组，数组名为 sh_a，有 10 个元素
```

对上面的数组，元素为从 a[0]到 a[9]，对数组赋值的方法，如 a[0]=0x05；也可以在定义的时候赋值，即初始化赋值。还可以定义多维数组，形如：short sh_a[n1][n2][n3][n4]；等等。在 C/C++中数组名代表地址，这和指针的意义是统一的，所以指针和数组名之间可以相互赋值。

```
struct   str_frame
{
      short   sh_fra1;
      long    lg_fra2;
      float   fl_res;
};
struct   str_frame   str_fra_data1;      //定义了一个结构体变量，变量名为 str_fra_data1
```

结构体是一种复合数据类型，它定义的变量中有三种不同性质的元素，它和其他的基本数据类型功能是一样的。对这种类型的变量的赋值方法，如 str_fra_data1.sh_fra1=0x04;。还可以定义指针或是数组，如 struct str_frame *str_fra_data2;，这时的赋值方法为：str_fra_data1 -> sh_fra1=0x04;。所有的 C 语句都以“;”结尾。

```
enum   True_False{false,true};    //定义一个枚举型变量，变量类型名为 True_False
```

枚举型变量是从 0 开始算起的，上面的 false 代表 0，true 代表 1。例如，定义一个枚举型的变量，如 enum True_False en_trueorfalse;，其赋值方法如 en_trueorfalse=false;，枚举定义中的枚举项目都是枚举常量，最多可以定义 2^{16} 个常量，所以可以直接用来赋值。

```
union   un_temp{ short i; long j; char k;} un_tmp_data1;   //定义一个共用体变量
```

共同体变量的优点在于节约存储空间。上例中，一个 short 型、一个 long 型和一个 char 型共同占有一个存储空间，所以在某一时刻只能存贮一种类型的数据。它所占用的地址空间为最长的那个变量所占用的地址空间。因此，un_tmp_data1 占用两个字。

现在将变量的定义方法总结一下：

变量类型说明符　　变量名;

变量类型说明符包括各种基本数据类型和扩展的数据类型；变量名在定义的同时可以赋初值。下面再举一个例子：

```
enum   en_bits{bit0,bit1,bit2,bit3};
union   un_addr{short i; long j; char k;};
struct   st_data
{
```

```
        short       sh_data[10];
        long        lg_data[100];
        float       fl_data[20];
        enum        en_bits     en_bits_val[10];
        union       un_addr     un_addr_val[10];
        struct      st_data     *pt_next;
};
struct   st_data *pt_st_data[10];
```

以上代码只是定义了一个指针数组变量 pt_st_data，这个变量包括一个 short 数组、一个 long 数组、一个 float 数组、一个枚举型数组、一个共同体数组和一个指针。虽然这个指针变量占有地址不大，但它指向的任一个变量将占有的地址数为 281×16 bits，显然这个庞然大物近期内可能不会出现在面向 DSP 的 C/C++程序设计中。在 C5000 系列中，一般在计算占用地址空间时以字为单位，一个字为 16 bits，而不是以字节为单位。

对变量的赋值采用“=”运算符。

下面给出一个简单的 C 程序示例：

```
main()
{
    short   sh_t1;
    short   sh_t2;
    short   sh_res;
    sh_t1=0x05;
    sh_t2=0x09;
    sh_res=sh_t1+sh_t2;

    while(1){};  //死循环等待。可以在观察窗口中观察变量的值，如图 3-4 所示
}
```

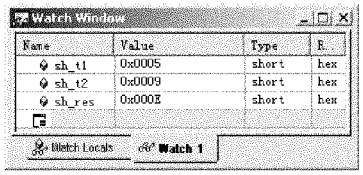


图 3-4　观察窗口

这个程序以及下面的示例程序的运行，都需要前面列出的 user_audio.cmd、vectors.asm 和 rts.lib 等文件的加入。

用户可以用“typedef”按自己的需要重定义一些常用的数据类型，如：

```
typedef   unsigned short    DATA;
```

语法：　typedef　　已有数据类型　　新数据类型

在上例中，DATA 与 unsigned short 是同一个涵义，所以：

```
unsigned    short   ush_table;
DATA   ush_table;
```

以上两个定义是等同的。

3) 常量与变量的对比

为了便于理解，将常量和变量做一个对比：映射存储器一旦建立，整个存储空间是被按序编址的。变量和常量都是指代一个数据存储单元，变量名和常量名指代了地址号。变

量名对应的地址内容是可以改变的，而常量对应的地址内容是不能改变的，必须在程序初始化时指定。一个完整的程序是离不开变量和常量的，常量最典型的应用就是构造各类查找表，可以固化到 ROM 中。用户只需在程序中定义变量和常量，.cmd 文件负责分配地址空间。变量只能占用 RAM 空间，常量可以占用 RAM 空间，也可以固化到 ROM 空间中。

需要补充说明的是，C/C++中可以定义全局的寄存器变量，借用关键字 register，且变量名只能是 AR1 或 AR6；在程序中使用了寄存器变量，应保证程序的其他部分不会占用这两个寄存器，否则会出现数据处理错误，而且很难查错。

3．C/C++地址变量

面向 DSP 的 C/C++程序设计中有一类专用的变量，这些变量是指针类型，负责 C5000 的片上外设的输入/输出数据管理，因为这些输入和输出地址在映射存储空间的地址是固定的，可以称之为地址变量。例如，对于 VC5402 来说，多通道缓冲串口 0（McBSP0）的接收数据寄存器 1(DRR10)在数据映射存储器中的映射地址固定为 21h，对这个地址的读操作等价于读 McBSP0 的 DRR10。

C/C++语句访问 DSP 片上外设唯一的方法就是借助这些外设的映射寄存器地址，访问定义常借助关键字 volatile 来表示，定义和访问方法如下：

```
volatile  short  *mcbsp0_drr10;
short  sh_drr10_data;
sh_drr10_data=*mcbsp0_drr10;
```

使用关键字 volatile 可以避免 C/C++的优化，使这些地址变量保存下来，专用于访问这些寄存器时使用。

C/C++中访问 DSP 的 I/O 空间的方法是借助关键字 ioport 来实现的。对于 C54xx 和 C55xx，其语法定义格式有所不同。

在 C54xx 中的格式为：

```
ioport   数据类型   porthex_num
```

其中，ioport是定义访问I/O空间的关键字。因为I/O空间在C54xx中只有64KW（W代表字），所以，数据类型只能为char、short、int、unsigned等16 bit的类型。对于访问I/O空间的地址100h，则变量名必须命名为port100。

```
ioport  short  port100;
short   sh_a,sh_b;
```

读操作：`sh_a = port100;`

写操作：`port100 = sh_b;`

作为函数调用时，采用的是传值方式，不是传址方式，即传送的是数值而不是地址。例如：

```
call_func(port100); //将 port100 内的数值传送给 call_func 函数
```

在 C55xx 中的格式为：

```
ioport  数据类型  *变量名;
```

下面以访问外部 I/O 空间的地址 100h 为例说明它的用法：

```
ioport  short  * io_tl16c55;
```

```
short   sh_addr;
short   sh_read_data;
sh_addr=100h;
io_tl16c55=&sh_addr;
sh_read_data=*io_tl16c55;
```

//C55xx 比 C54xx 灵活一些，而且可以访问更多的 I/O 地址空间

C55xx 中还比 C54xx 多了一个关键字 onchip 和另一个关键字 restrict。使用关键字 onchip 进一步定义的变量只能存储在片上存储区间中，不能存储在外部映射来的存储区域上，它同时指明这个变量可能会参与 CPU 的乘、加运算，如 onchip short sh_macdata1;。关键字 restrict 进一步声明的变量一般位于函数中，它限制了这些存储空间不能被其他的变量所覆盖，也不能被其他函数所访问，如 restrict short sh_func_a1;。

4．C/C++数据操作

掌握了 C/C++的数据类型、变量、常量的知识之后，需要进一步明白在程序中如何使用这些变量和常量，即要明白 C/C++的运算符、表达式和语句。其中语句将在第 3.4.2 节中集中讲述，在这里仅是提及它。

前面提到了 C/C++是一种函数式的语言，C/C++程序是由一个主函数和零个或多个子函数组成的，主函数中一定有“死循环”负责 DSP 的数据处理。每个函数又是由零个或多个 C/C++语句组成，每个语句都以“;”号结尾。不管是定义变量、常量，还是表达式运算等等，只有以分号结尾，C/C++才认为是合法的可执行语句。下面举个简单的实例：

```
void   q152fl(short   *, float   *,short);   //函数声明语句
main()                                  //主函数
{
    short   sh_org[100];                //定义变量语句
    float   fl_des[100];                //同上
    q152fl(sh_org,fl_des,100);          //调用函数语句
    while(1)
    {}
}

void   q152fl(short *sh_sou,float *fl_cha,short   sh_len)        //子函数
{
    //子函数内容
}
```

将变量或常量经过运算符连接起来就构成了表达式，表达式加上分号即为语句。

下面列出了常用的运算符：

? 赋值运算符：　=

? 数学运算符：　+（加）、-（减）、　*（乘）、/（除）、%（取模）

在 C5000 系列中，取模按分子的符号决定取值的符号，如 10%-3=1; -10%3=-1;等。

上面两种运算符可以结合使用，如 +=，-=，/=，*=，%=等；还可以有++、--等运算符。示例及意义如下：

```
short a,b,c;
a=0x08;
b=0x09;
c=0x00;
b+=a;     等价于 b=b+a;
c++;      等价于  c=c+1;
++c;      等价于  c=c+1;
```

上面两式的区别，用例子表示如下：

```
a=0x05;
c=0x05;
a=a+(c++);   // 运算完成后，a=0x0A，c=0x06，c 先参与表达式运算再加上 1
a=a+(++c);   // 运算完成后，a=0x0B，c=0x06，c 先加上 1 之后再参与表达式计算
```

++、--常被称为自增、自减运算符。

在表达式中，只能使用小括号，不能使用其他括号。括号内的运算优先级最高，其次是先乘除后加减的次序。

? 关系运算符：C/C++中认为 0 为假，非 0 或 1 为真。对于表达式的结果，只有 1 代表真。对于表达式的输入，一切非 0 视为真。关系运算符及示例如表 3-4 所示。

表 3-4 关系运算符一栏

运算符含义	运算符	示　例	值
等于	= =	0x08= =0x0A	0
不等于	! =	0x08 != 0x0A	1
大于	>	0x0A>0x08	1
大于或等于	>=	0x0A>=0x08	1
小于	<	0x0A<0x08	0
小于或等于	<=	0x08<=0x08	1

? 逻辑运算符：逻辑运算符只有三个，用以进行表达式运算结果之间的逻辑运算，如表 3-5 所示。

表 3-5 逻辑运算符一栏

运算符含义	运算符	示　例	值
逻辑与	&&	(x= =0x05)&&(y= =0x06)	1
逻辑或	‖	(x= =0x05)‖(y!=0x06)	1
逻辑非	!	!(y= =0x06)	0

注：假定 x=0x05，y=0x06

？运算符的优先级自高向低排序为：

逻辑非（!），算术运算符，关系运算符，逻辑运算符&&、‖，赋值运算符。

？C/C++中的位运算符包括：

&（按位与）、|（按位或）、^（按位异或）、~（取反）、<<（左移）、>>（右移）

位运算符在 C5000 的编程中得到了广泛的应用。它只能对整型和字符型数据进行处理。

？三元运算符：C/C++中唯一的一个三元运算符是“？：”，它有三个参加运算的元素，而且有返回值。调用格式为：

(exp1) ? (val1) : (val2)

其功能是先判断表达式 exp1 的真假，为真时返回表达式 val1 的值，为假时返加表达式 val2 的值。示例如下：

```
short sh_res;
short sh_a,sh_b;
sh_a=0x08;sh_b=0x09;
sh_res=(sh_a>sh_b)?sh_a:sh_b;
```

其运算结果为：sh_res=0x09，即为求两个数的最大值。

？sizeof 运算符：sizeof 运算符用于求一个变量或常量占有存储空间的字数。这与普通的 C/C++中的有所区别，在通用的 C/C++中这个运算符是计算占有存储空间的字节数。示例如下：

```
short   sh_size1;
sh_size1=sizeof(sh_size1);    //或
sh_size1=sizeof(0x08);
```

？逗号运算符：使用“,”连接起来的表达式，按从左向右的次序依次计算各个表达式的值，整个逗号表达式的值为最后一个表达式的值。示例如下：

```
short   sh_t1;
short   sh_t2;
sh_t1=0x05;
sh_t2=0x09;
sh_t1=((sh_t1+sh_t2),sizeof(0x04));
```

其运算后的结果为 sh_t1=0x01。

？数组下标运算符：数组下标运算符用“[ ]”表示。通过这个运算符可以访问数组中的任意位置的元素。示例如下：

```
short   sh_a[10];    //定义一个具有 10 个元素的数组，从 sh_a[0]至 sh_a[9]
sh_a[8]=0x08;       //对第 8 个元素赋值
```

？结构/联合成员运算符：这个运算符是“.”，也用于 C++的对象成员访问中。示例如下：

```
struct   st_tabl
{
    short   sh_ord;
    float   fl_val;
```

```
};
struct  st_tabl  st_tabl_list;
st_tabl_list.sh_ord=0x01;
st_tabl_lish.fl_val=0.7329;        //赋值
```

? 结构/联合指针运算符：这个运算符“->”也可用于 C++指针对象中，示例如下：

```
struct  st_tabl
{
    short  sh_ord;
    float  fl_val;
};
struct  st_tabl  *st_tabl_list;
```

对其中元素的调用为：st_tabl_list -> sh_ord，但不能显式赋值，如下式是错误的:

```
st_tabl_lish -> fl_val=0.7329;        //错误的赋值方式
```

? 引用运算符、地址运算符以及指针取值运算符：只有 C++支持引用运算符；C/C++都支持地址运算符，这两个运算符都用“&”表示；指针取值运算符用“*”表示。引用运算符是用来产生已有变量的一个别名，对这个别名的一切操作和修改都是对原变量的操作和修改。地址运算符是求一个变量的地址；而指针取值运算符是求一个地址内的变量的值，正好与地址运算符的作用相逆。示例如下：

```
short  sh_org;
short  &sh_org_ref=sh_org;        //引用
sh_org_ref=0x08;
```

上面三个语句的运算结果为：sh_org=0x08。

```
short  sh_org;
short  *sh_addr;
short  sh_val;
sh_org=0x08;
sh_addr=&sh_org;
sh_val=*(&sh_org);
```

上述语句是将变量 sh_org 的地址送给指针 sh_addr，将 sh_org 的值送给 sh_val。为了说明这些运算符的用法，这里故意走了一个弯路。

? 强制类型转换运算符：C/C++中的强制类型转换运算符就是把数据类型当作运算符的一种表示方法。下面举例说明：

```
short  sh_val;
sh_val=(short)12.345;   //C/C++均支持这种表示方法的类型转换
sh_val=short(12.345);   //只有 C++才支持这种表达式方法
```

在 C5000 系列中，几乎所有类型之间是可以相互进行强制类型转换的。这种方法在面向 DSP 的 C/C++程序设计中较常用。

? new 和 delete 运算符：这两个运算符是 C++特有的，new 运算符用来给变量开辟一些存储空间，delete 用来将这些空间释放掉。示例如下：

```
short    *sh_a=new short[10];
delete    sh_a;
```

这两个运算符应联合使用，在程序运行过程中可以用 new 开辟出新的存储空间，在程序结束前应用 delete 释放这些存储空间。

3.4.2　C/C++控制语句

C/C++中的语句组有三种执行方式：顺序、分支和循环。下面我们将分别介绍这三种方式。无论功能如何强大的 C/C++程序，在程序流程上只有这三种方式。语句是函数的基本单元，函数又是 C/C++程序的组成单元，因此，要设计好程序，首先应把如何编写 C/C++语言语句掌握好。

1．C/C++顺序语句

C/C++顺序语句是最基本的一种执行方式，也是 C/C++程序总体的一种执行方式，如图 3-5 所示。这是一个语句执行完成后再去执行另一个语句的运行方式，或是一个语句组执行完毕后再去执行下一个语句组的运行方式。这种运行方式简单，容易被人接受和理解。下面举一个例子加以说明。

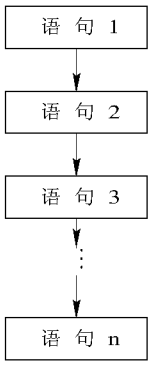


图 3-5　顺序语句

```
#include   "myallinc.hpp"
short   sh_in_data[128];
short   sh_out_data;
void   main(void)
{
    init_brd();    //系统初始化
    while(1)
    {
        data_gen(sh_in_data,128);          //调用一个函数，从端口读入 128 点的一帧数据
        myfir(sh_out_data,sh_in_data,128);
        //调用一个函数，作 FIR 滤波处理，结果存入 sh_out_data 中
        data_out_phe(sh_out_data);         //将数据送输出缓冲区
    }
}
```

上面这个程序中，while(1)内的各语句是顺序执行的（这个程序没有给出调用函数的代码，不能上机实习）。

C/C++中，语句组是用“{}”括起来的一组语句。

2．C/C++分支语句

C/C++分支语句是指需要通过判断关系式或逻辑表达式的值，才能决定下一步进行什么操作的语句，这种结构如图 3-6 所示。在 C/C++中有两种实现分支（或选择）的语句：if 语句和 switch 语句。现分别介绍如下：

if 语句的语法及示例：

if(表达式) {语句组 1}

例如：

```
short   sh_a,sh_b,sh_c;
sh_a=0x08;
sh_b=0x09;
sh_c=sh_a;
if(sh_b>sh_a)sh_c=sh_b;
```

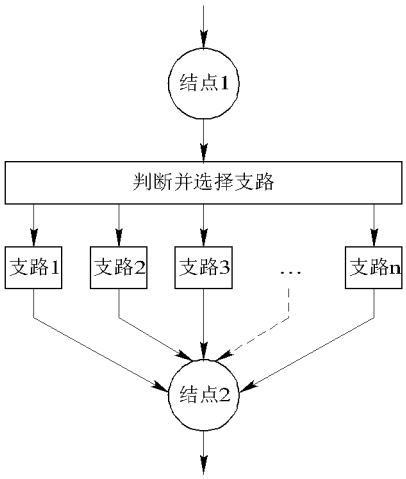


图 3-6　分支结构

```
if(表达式)
{语句组 1}
else
{语句组 2}
```

例如：

```
if(sh_a>sh_b)
    sh_c=sh_a;
else
    sh_c=sh_b;
```

当只有一个语句时，可以不用加“{}”。但为了避免歧义，建议都加“{}”。例如：

```
if(表达式 1){语句组 1}
else if(表达式 2){语句组 2}
else if(表达式 3){语句组 3}
else if(表达式 4){语句组 4}
…
else if(表达式 n){语句组 n}
else{语句组 n+1}
```

示例如下：

```
if(sh_a>sh_b)
{
    sh_c=1;
}
else if(sh_a= =sh_b)
{
    sh_c=0;
}
else
{
    sh_c=-1;
}
```

if 语句可以嵌套，即上面的语句组又可以是 if 语句。具有代表性的二级嵌套语法如下：

```
if(表达式 1)
{
    if(表达式 11){语句组 11}
    else if(表达式 21){语句组 21}
    else if(表达式 31){语句组 31}
    else if(表达式 41){语句组 41}
    …
    else if(表达式 n1){语句组 n1}
    else{语句组 n1+1}
}
else if(表达式 2)
{
    if(表达式 12){语句组 12}
    else if(表达式 22){语句组 22}
    else if(表达式 32){语句组 32}
    else if(表达式 42){语句组 42}
    …
    else if(表达式 n2){语句组 n2}
    else{语句组 n2+1}
}
else if(表达式 3){语句组 3}
else if(表达式 4){语句组 4}
…
else if(表达式 n){语句组 n}
else{语句组 n+1}
```

上面仅给出了部分二级 if 嵌套，还可以有多级 if 嵌套，方法类似。

switch 语句可以实现与 if 语句相同的功能，可以根据实际情况选用。

switch 语句的语法及示例如下：

```
switch(表达式)
{
    case  常量表达式 1：语句组 1
    case  常量表达式 2：语句组 2
    …
    case  常量表达式 n：语句组 n
    default：语句组 n+1
}
```

当表达式的值为某一常量表达式的值时，程序就会跳转到相应的语句组去。如果执行完这个语句组后，想跳出 switch 语句，则需在语句组后面加上 break；如果还想让它继续执

行后面的语句，则不用加入 break。当所有的情况不成立时，执行 default 语句组。注意：“break;”语句也用于跳出循环体外，执行循环体外的下一条语句。

示例如下：

```
main()
{
    short   sh_t1;
    short   sh_t2;
    sh_t1=0x05;
    switch(sh_t1)
    {
        case 0x05:
            sh_t2=1;
            break;
        case 0x06:
            sh_t2=0;
            break;
        default:
            sh_t2=-1;
    }
    while(1){};
}
```

这个例子的运行结果为 sh_t2=0x01。

3．C/C++循环语句

C/C++用于实现程序循环的方法有三种，它们都可以用来实现死循环，而且循环是可以嵌套的。下面仅介绍了基本循环，没有具体介绍循环的嵌套，循环的嵌套只是将循环中的语句组用一个循环体来代替即可，C/C++支持多级循环嵌套。循环结构如图 3-7 所示。

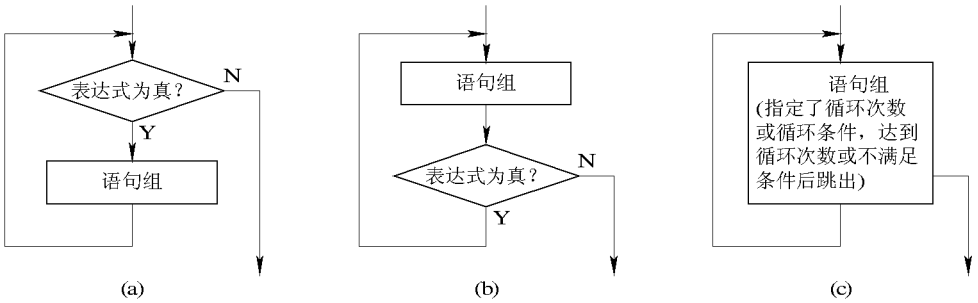


图 3-7　三种循环模式

(a)　当型循环; (b)　直到型循环; (c)　指定次数或条件的循环

1) goto 循环

goto 循环的一般格式示例如下：

```
g_Label:        //指定一个标号
    语句组;
goto g_Label;
```

死循环的方法：

```
g_L1:
{
    语句组；
}
goto g_L1;
```

跳出循环的方法：

```
g_L1:
    语句组 1；
    if(条件表达式)break;
    语句组 2;
goto g_L1;
```

另一个关键字 continue 与 break 功能相似。但是，continue 是结束本次循环，即 continue 下面的语句组本次循环不执行了，而从循环头部再继续执行循环；break 则是结束循环体，进入循环体下面的语句。

使用 goto 循环注意不要把跳转标号弄乱。这种循环不是结构化的设计，常常不使用。

2) while 循环

while 循环的语法如下：

```
while(表达式)                    //先判断后执行
{
    语句组；
}
```

死循环的方法如下：

```
 while(1)
{
    语句组；
}
```

跳出循环的方法是当表达式为假时，会跳出循环体。当然也可以在循环体中加入一些跳出条件和 break 语句，或加入 continue 语句，每次循环时根据条件忽略掉一些语句的执行。

```
Do                              //先执行后判断
{
    语句组；
}while(表达式);
```

死循环的方法如下：

```
do
{
    语句组；
}while(1);
```

当表达式为假时，跳出循环。也可以在循环体中加入条件跳出语句。

while 循环是一种常用的循环，而且几乎是每个 C/C++程序都不可少的一种循环。

3) for 循环

for 循环的语法如下：

```
for(循环表达式 1；循环条件；循环表达式 2)
{
    语句组；
}
```

循环表达式 1 在开始 for 循环时计算，循环表达式 2 从第二次循环起每次循环时都计算，循环条件在每次循环时都判断。

一个简单的算法示例如下：

```
short   sh_sum;             //计算 1+2+…+100=?
unsigned short   ush_i;
sh_sum=0x00;
for(ush_i=1;ush_i<=100;ush_i++)
{
    sh_sum+=ush_i;
}
```

死循环的示例如下：

```
for(;;)    //注意两个“；”号是不能少的
{
    语句组；
}
```

跳出循环的方法：可以通过循环条件跳出循环，或是在循环体内部加入判断条件。

在 for 循环(循环表达式 1；循环条件；循环表达式 2)中，循环表达式 1、循环条件、循环表达式 2 都是可以省略的，但是两个“；”号不能省略。读者可视自己程序的需要，灵活运用。如下例：

```
short   sh_sum;             //计算 1+2+…+100=?
unsigned short   ush_i;
sum=0x00;
ush_i=1;
for(;;)
{
```

```
        if(ush_i<=100)
            sh_sum+=ush_i;
        else
            break;
        ush_i++;
    }
```

上例中，将循环表达式 1 放到了循环体前面，循环判断条件和循环表达式 2 放在了循环体内部。for 循环也可以多级嵌套。

3.5　C/C++语言函数

进行模块化的软件设计也是面向 DSP 的 C/C++程序设计的一条基本原则。按所要实现的功能将一个程序分成很多模块，在 C/C++中每个模块就是一个函数，或者把几个模块组成一个函数。函数是 C/C++进行模块化设计的重要手段。可以将一个或几个函数存入一个文件中，在主程序头部用“#include ”文件名”” 将其包括在内，即可以使用这些函数；或者把一些函数放在主程序文件的后面，在主程序前应加入这些函数的声明；也可以直接将这些函数放在主程序前面。CCStudio 采用工程编译连接方法，所以也可以用源文件的形式加入到工程中，一起编译，程序会自动连接成一个可执行文件。注意，将函数的源代码写入到其他的.c 或.cpp 文件中，在主程序文件中调用这些函数时，应在声明要调用的函数名前加上关键字 extern。我们将在第 3.5.2 节给出这种方法的完整实例。

下面首先介绍编写函数的方法，接着介绍 C/C++语言的库函数和 CCStudio 提供的专用 DSPLIB 库函数。这些专用库函数是用汇编语言编写的，执行速度快，在进行实际 DSP 程序设计时，应尽量使用这些专用库函数。

3.5.1　C/C++自定义函数

C/C++函数由函数头部和函数体组成。函数头部常被复制到主函数前作为函数的声明，即函数原型，告诉编译器该函数的主体在主程序后面定义了。语法形式如下：

```
返回类型　函数名(形式参数列表)
{
        函数体;
}
```

返回类型可以为基本数据类型，也可以为扩展的数据类型或是空类型。当函数有返回值时，使用 return 语句。函数名的指定与变量名的指定相似。形式参数表可以为空，也可以有多个，用来向函数体内部传递数据或地址。形式参数和函数体内部定义的变量均为局部变量，它们的作用域仅限于函数体内部；而定义于主函数的变量或主函数体外的变量一般可以称为全局变量，它们的作用域是从开始定义到主程序结束。

调用函数时，必须用实际的数据变量代替形式参数（形参），这些实际的变量称为实参。实参对形参的传递方式有两种：一种是传值方式，另一种是传址方式。在传值方式中，实

参的值会传递给形参，形参在函数体内参与运算受到改变时不会影响到实参的值，这时实参和形参是占有完全不同的地址空间；在传址方式中，实参的地址传给了形参，这时对于形参的改变会影响到实参的改变，这种方法在 C++中也可以采用引用方式来实现。简单地说，就是在传值方式中，实参传给形参值是可以的，形参是不能向实参传值的；而在传址或是引用方式中，实参的值可以传给形参，形参的值也可以传给实参。应根据情况的不同采用不同的实参到形参的传递方式。

下面以一个简单的例子说明一个函数的定义方法及实参到形参的传递方式。

```
void   sh_swap1(short sh_x,short sh_y)
{
      short sh_tm;
      sh_tm=sh_x;
      sh_x=sh_y;
      sh_y=sh_tm;
}
void   sh_swap2(short *sh_x,short *sh_y)
{
      short sh_tm;
      sh_tm=*sh_x;
      *sh_x=*sh_y;
      *sh_y=sh_tm;
}
void   sh_swap3(short &sh_x,short &sh_y)
{
      short sh_tm;
      sh_tm=sh_x;
      sh_x=sh_y;
      sh_y=sh_tm;
}

int main(void)
{
      short sh_a1,sh_b1,sh_a2,sh_b2,sh_a3,sh_b3;
      sh_a1=sh_a2=sh_a3=0x07;
      sh_b1=sh_b2=sh_b3=0x09;
      sh_swap1(sh_a1,sh_b1);
      sh_swap2(&sh_a2,&sh_b2);
      sh_swap3(sh_a3,sh_b3);

      while(1){};
}
```

运行结果如图 3-8 所示。

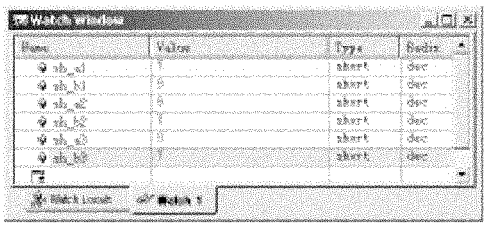


图 3-8　运行结果

从运行结果可以看出，sh_swap1 是传值的，形参的改变不影响实参；sh_swap2 是传址的，形参的交换也使实参的值交换了；sh_swap3 是引用的方式(C++中才有的)，形参的交换使得实参的值也交换了。

从上例可以看出，自定义的函数是为了完成一个特定的功能，使程序的思路更加清晰，增强了程序的模块化思想。应当充分利用自定义的函数库，并做出详细的程序功能和输入/输出说明，准备详细的编程文档。不要将许多功能都写在一个函数中，然后再将这个函数放在主程序中，这样的程序是很难有可扩充性和借鉴性的。应尽量使用不同的函数来定义不同的功能，并且要保持各个函数的功能的独立性和接口的鲁棒性，这样，函数移植起来很方便，其重用性也会很强。

C/C++支持函数递归调用，即一个函数可以调用它本身。下面举一个实例：

```
short sh_fib(short sh_rabit)
{
    if(sh_rabit<3)
        return 1;
    else
        return(sh_fib(sh_rabit-2)+sh_fib(sh_rabit-1));
}
int main(void)
{
    short sh_rab;
    sh_rab=sh_fib(7);

    while(1){};
}
```

3.5.2　C++函数重载

函数重载只能在 C++语言中实现，C 语言则无能为力。所谓函数重载是指在 C++中可以存在同名的函数，这些函数的返回值或是形式参数是不相同的。C++在调用同名函数时，靠其不同的返回值或参数来区分。下面给出一个简单的实例（这个实例需要 vectors.asm 和 user_audio.cmd 的支持，并加入 rts.lib 库，因为改变了 CLKMD1、CLKMD2 和 CLKMD3 的配置，可以调用 SY-5402EVM 板的 GEL 函数来初始化 EVM 板，即使不初始化也没有关系）。

下面两个文件也需要加入到工程文件中。

```
//filename:cadd_max.cpp
int maxiz(int a1,int a2)
{
    int z1;
    z1=a1;
    if(a1<a2)
```

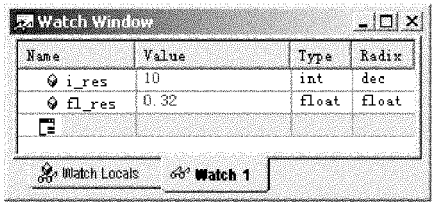
```
        z1=a2;
    return z1;
}
float maxiz(float a1,float a2)
{
    float z1;
    z1=a1;
    if(a1<a2)
        z1=a2;
    return z1;
}

//filename: cadd_test.cpp
extern int maxiz(int,int);
extern float maxiz(float,float);

int main(void)
{

    int i_a1,i_a2,i_res;
    float fl_a1,fl_a2,fl_res;
    i_a1=5;
    i_a2=10;
    fl_a1=0.2;
    fl_a2=0.32;
    i_res=maxiz(i_a1,i_a2);
    fl_res=maxiz(fl_a1,fl_a2);

    while(1){};
}
```



执行结果如图 3-9 所示。

图 3-9　执行结果

此外，C++还可以对大部分运算符重载。下面给出一段运算符重载用法的实例，其中用到了关键字 operator。运算符 new 和 delete 也是可以重载的。

```
struct st_add
{
    short sh_add1;
    float fl_add2;
};
```

```
struct st_add operator +(struct st_add st_num1,struct st_add st_num2)
{
    struct st_add st_res;
    st_res.sh_add1=st_num1.sh_add1+st_num2.sh_add1;
    st_res.fl_add2=st_num1.fl_add2+st_num2.fl_add2;
    return st_res;
}

int main(void)
{
    struct st_add st_dig1,st_dig2,st_fin;
    st_dig1.sh_add1=0x01;
    st_dig1.fl_add2 =2.1;
    st_dig2.sh_add1 =0x09;
    st_dig2.fl_add2 =3.2;
    st_fin=st_dig1+st_dig2;
    while(1){};
}
```

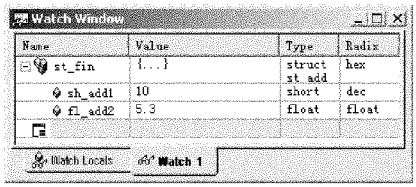


图 3-10　运行结果

该程序的运行结果如图 3-10 所示。显然是将运算符加号重载了。

C/C++都支持内联函数，这种类型的函数以 inline 开头，编译器可直接将这部分代码合并到主程序中去。函数原型示例如下：

```
inline short sh_data_pro(short *,short *,short);
```

建议在面向 DSP 的 C/C++语言程序设计中多使用内联函数。

3.5.3　中断函数

C/C++中定义中断函数的方法，是使用关键字 interrupt。中断函数在 C5000 系列程序设计中几乎是必不可少的。中断函数无返回值，也没有参数。下面介绍中断函数的定义和调用方法。

中断函数的定义如下：

```
interrupt   void   inte_INT0()
{
    //中断函数内容
}
```

调用中断函数时，系统会完成 CPU 寄存器的入栈保存，中断返回后，系统会完成 CPU 寄存器的出栈恢复。在 C/C++中，用户可以专心设计其想实现的中断功能。

中断调用由中断向量表负责完成，示例如下。

(1) 在 C 语言中调用中断示例：

```
        .    sect ".vectors"
             .ref _inte_INT0      //引用外部的中断函数符号
             .ref _c_int00

        .    align   0x80

RESET:
             BD _c_int00
             STM #128,SP
nmi:         RETE
               NOP
               NOP
               NOP
sint17  .space   4*16
sint18  .space   4*16
sint19  .space   4*16
sint20  .space   4*16
sint21  .space   4*16
sint22  .space   4*16
sint23  .space   4*16
sint24  .space   4*16
sint25  .space   4*16
sint26  .space   4*16
sint27  .space   4*16
sint28  .space   4*16
sint29  .space   4*16
sint30  .space   4*16

int0:        B     _inte_INT0 子   //在 INT0 中断触发时，跳转去执行中断服务程序
               NOP
               RETE
int1:        RETE
               NOP
               NOP
               NOP
int2:        RETE
               NOP
               NOP
               NOP
```

```
tint0:      RETE
            NOP
            NOP
            NOP
brint0:     RETE
            NOP
            NOP
            NOP
bxint0:     RETE
            NOP
            NOP
            NOP
brint1:     RETE
            NOP
            NOP
            NOP
bxint1:     RETE
            NOP
            NOP
            NOP
            .end
```

(2) 在 C++语言中调用中断示例：

```
            .sect ".vectors"

            .ref _c_int00
            .ref _inte_INT0__Fv        //C++语言中的中断符号
      .     align   0x80

RESET:
            BD _c_int00
            STM #128,SP
nmi:        RETE
              NOP
              NOP
              NOP
sint17   .space   4*16
sint18   .space   4*16
sint19   .space   4*16
```

```
sint20  .space  4*16
sint21  .space  4*16
sint22  .space  4*16
sint23  .space  4*16
sint24  .space  4*16
sint25  .space  4*16
sint26  .space  4*16
sint27  .space  4*16
sint28  .space  4*16
sint29  .space  4*16
sint30  .space  4*16

int0:     B     _inte_INT0__Fv      //INT0 中断发生时去执行该中断
          NOP
          RETE
int1:     RETE
          NOP
          NOP
          NOP
int2:     RETE
          NOP
          NOP
          NOP
tint0:    RETE
          NOP
          NOP
          NOP
brint0:   RETE
          NOP
          NOP
          NOP
bxint0:   RETE
          NOP
          NOP
          NOP
brint1:   RETE
          NOP
          NOP
          NOP
```

```
bxint1:    RETE
           NOP
           NOP
           NOP
           .end
```

细心观察可以发现上面两种调用的不同之处：在 C 中，中断向量的标号只是加了一个下划线“_”；而在 C++中，还加了一个“__Fv”后缀。这个中断向量的标号可以从 MAP 表文件中找到。

3.5.4　C/C++库函数

假设读者将 CCStudio 安装在 c:\ti 目录下，这是默认安装目录，则 c:\ti\c5400\cgtools\include 和 c:\ti\c5500\cgtools\include 两个目录下的文件即为 C/C++的库函数文件（其实 C5400 和 C5500 具有相对独立的编译支持环境，它们只是共用 CCStudio 集成环境而已）。可以将那些没有扩展名的文件加上.hpp 扩展名（这些文件是用于 C++下的，不知为什么，CCStudio 中没有给这些文件定义扩展名）。加上.hpp 扩展名后，两个目录的文件如下所示。

(1) c:\ti\c5500\cgtools\include 目录下的文件：

具有.h 的文件有 38 个，分别为：access.h， arith.h， assert.h，c55x.h，ctype.h，errno.h，extaddr.h，file.h，float.h，formi32.h，frcaddd.h，frcdivd.h，frcmpy.h，gsm.h，ieeed.h，ieeemask.h，intrindefs.h， iso646.h，ld3_32.h，limits.h， linkage.h， math.h，mathf.h， mathl.h，numconst.h，reald.h，realdi.h，renormd.h，setjmp.h，stdarg.h，stddef.h，stdio.h， stdlib.h，string.h，syntd.h，syntdi.h，time.h， unaccess.h；

具有.hpp 的文件有 19 个，分别为：cassert.hpp， cctype.hpp， cerrno.hpp， cfloat.hpp，climits.hpp， cmath.hpp， cmathf.hpp， cmathl.hpp， csetjmp.hpp， cstdarg.hpp， cstddef.hpp，exception.hpp，new.hpp， cstdlib.hpp， cstdio.hpp， cstring.hpp， stdexcept.hpp， ctime.hpp，typeinfo.hpp。

(2) c:\ti\c5400\cgtools\include 目录下的文件：

具有.h 的文件有 19 个，分别为：access.h， assert.h， ctype.h， errno.h， file.h， float.h，intrindefs.h， ldcio.h， limits.h， linkage.h， math.h， setjmp.h， stdarg.h， stddef.h， stdio.h，stdlib.h， string.h， time.h， unaccess.h；

具有.hpp 的文件也有 19 个，分别为：cassert.hpp， cctype.hpp， cerrno.hpp， cfloat.hpp，climits.hpp，cmath.hpp， cmathf.hpp， cmathl.hpp， csetjmp.hpp， cstdarg.hpp， cstddef.hpp，exception.hpp， new.hpp， cstdlib.hpp， cstdio.h， cstring.h， stdexcept.hpp， ctime.hpp，typeinfo.hpp。

许多库文件只是辅助性的。下面仅就 C5400 系列介绍几个常用的重要库文件，使读者知道如何去阅读和运用库文件中的函数声明，这样，对 C5500 系列就不难理解了。C5500 系列的库文件比 C5400 系列要多一些，即使同名的库文件，内容也不一定完全相同。C5500 系列的专有库文件和一些C5400 系列没有的库函数在C5400 系列上没有相应的运行支撑库，所以即使可以调用也是不能连接成可执行文件的，即 C5400 和 C5500 的编译系统是严格分

开的，支撑库也是分开的。上面加了阴影的库文件是 C5400 和 C5500 同名的文件，其内容有些是不同的。C 语言的库文件 C++均可调用，C 库函数是 C++库函数的一个真子集。C5400 和 C5500 的 C++所有库文件名是相同的，都是 19 个，但编译系统是严格分开的。应特别强调，C++支持 C 的所有库函数。调用这些库文件中的内容时，应在程序中加入“#include　“库文件名”” 或“#include　<库文件名>”。“< >”与双引号的区别在于编译器寻找库文件的路径是不相同的，建议将一些自定义函数与程序文件放在同一目录或一个专用目录下。调用函数时应使实参与形参同属于一种数据类型，当然也可以使用强制类型转换。在库文件中广泛地使用了预处理命令，就是前面加了“#”号的指令。如下面的条件编译指令：

```
#if　常量表达式　　//不满足要求的语句组不会参加编译
    语句组
    #ifdef　宏名　　//如果定义这个宏名，将执行下面的语句
        语句组
#else
    语句组
#endif
```

打开一个库文件，将光标移动到所需要的函数名上，按下 F1 键，即可以得到相应的“帮助”。下面介绍 C5400 的常用库函数，这些函数定义在上面列举的库文件中，它们的原型及函数体大都位于文件 rts.src 中。

库文件名：ctype.h 或 cctype.hpp

作用：字符函数，如表 3-5 所示。

表 3-5　字符函数的原型及含义

函 数 原 型	含　　义
int isalnum(int c);	当输入的变量是 ASCII 码或数字时返回真，否则返回 0
int isalpha(int c);	当输入的变量为 ASCII 码时返回 1，否则返回 0
int isascii(int c);	当输入的变量为 ASCII 码（从 0 到 127）时返回 1，否则返回 0
int iscntrl(int c);	当输入的变量为控制符时（ASCII 码的 0～31 和 127）返回 1，否则返回 0
int isdigit(int c);	当输入的变量为 0～9 的数字时返回 1，否则返回 0
int isgraph(int c);	当输入的变量为非空格字符时返回 1，否则返回 0
int islower(int c);	当输入小写 ASCII 码时返回 1，否则返回 0
int isprint(int c);	当输入的变量为打印字符时（ASCII 的 32～126）返回 1，否则返回 0
int ispunct(int c);	当输入的变量为 ASCII 标点符号时返回 1，否则返回 0
int isspace(int c);	判断输入为空格、回车、Tab 键、换行等时返回 1，否则返回 0
int isupper(int c);	判断为大写 ASCII 字母时返回 1，否则返回 0
int isxdigit(int c);	判断为十六进制数时返回 1，否则返回 0
char toascii(int c);	将 int 型的变量转换为一个合法的 ASCII 码字符
char tolower(int c);	如果输入为大写字母时转化为小写字母
char toupper(int c);	如果输入为小写字母时转化为大写字母

库文件名：limits.h 或 climits.hpp

作用：定义 signed char、char、short、int、long 型数据类型的最小值和最大值。注意在C/C++中表示无符号数的方法，如 65535u。

库文件名：assert.h 或 cassert.hpp

用途：出错信息。例如

void assert(int expr);

如果 expr 表达式为真，该函数无动作；如果为假时，它将向标准输出设备报告错误信息，并中止程序。

库文件名：math.h 或 cmath.hpp、cmathf.hpp、cmathl.hpp

作用：数学函数，如表 3-6 所示。

表 3-6　数学函数原型及含义

函　数　原　型	含　　义
double acos(double x);	计算反余弦函数
double asin(double x);	计算反正弦函数
double atan(double x);	计算反正切函数
double atan2(double y, double x);	计算 y/x 的反正切函数
double ceil(double x);	返回不小于 x 的浮点形式表示的整数
double cos(double x);	计算余弦函数
double cosh(double x);	计算双曲余弦函数
double exp(double x);	计算 e^x
double fabs(double x);	计算绝对值
double floor(double x);	返回一个用浮点数表示的不大于 x 的整数
double fmod(double x, double y);	返回 x/y 的余数
double frexp(double value, int *exp);	将 value 分解为一个小数部分和一个 2^{exp} 的乘积的形式，返回小数部分，exp 为传址方式
double ldexp(double x, int exp);	返回 value*(2^{exp})
double log(double x);	返回 ln(x)
double log10(double x);	返回 $\log_{10}(x)$
double modf(double value, double *iptr);	将一个浮点数分成浮点形式表示的整数和小数部分，返加小数部分
double pow(double x, double y);	返回 x^y
double sin(double x);	计算正弦函数
double sinh(double x);	计算双曲正弦函数
double sqrt(double x);	计算平方根
double tan(double x);	计算正切函数
double tanh(double x);	计算双曲正切函数

库文件名：stdlib.h 或 cstdlib.hpp

用途：通用函数库，列于表 3-7 中。

表 3-7 通用函数列表

函 数 原 型	含 义
void abort(void);	终止正在运行的程序
int abs(int i);	求绝对值
int atexit(void (*func)(void));	在程序终止时调用 func 指向的函数
double atof(const char *st);	将字符串转化为浮点数值
int atoi(const char *st);	将字符串转化为整数值
long atol(const char *st);	将字符串转化为长整数值
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	在 nmemb 数组中查找 key 指针指向的元素
void *calloc(size_t num, size_t size);	为 num 个元素分配存储空间, 每个元素占 size 个字
typedef struct { int quot; /* 商 */ int rem; /* 余数 */ } div_t; div_t div(int numer, int denom);	整数除法，返回商和余数
void exit(int status);	终止一个程序
void free(void *ptr);	释放由 malloc、calloc 和 realloc 分配的存储空间
char *getenv(const char *_string)	返回字符串的环境信息
long labs(long i);	返回长整形数的绝对值
typedef struct { long int quot; /* 商 */ long int rem; /* 余数 */ } ldiv_t; ldiv_t ldiv(long numer, long denom);	长整数除法，返回商和余数
int ltoa(long val, char *buffer);	把一个长整数值转变为一个字符串
void *malloc(size_t size);	为变量分配 size 个字的存储空间
void minit(void);	释放所有被 malloc、calloc 和 realloc 分配的空间
void qsort(void *base, size_t nmemb, size_t size, int (*compar) ());	按升序将 nmemb 数组进行排序
int rand(void);	返回一个从 0 到 RAND_MAX 之间的随机数，对于 16 bit 的 short 型，RAND_MAX=32767
void *realloc(void *ptr, size_t size);	改变 allocate 分配的存储空间大小
void srand(unsigned seed);	复位随机数发生器
double strtod(const char *st, char **endptr);	将一个字符串转换为一个浮点数
long strtol(const char *st, char **endptr, int base);	将一个字符串转换为一个长整数
unsigned long strtoul(const char *st, char **endptr, int base);	将一个字符串转换为一个无符号的长整数

库文件名：setjmp.h 或 csetjmp.hpp
用途：全局跳转，如表 3-8 所示。

表 3-8　全局跳转函数

函　数　原　型	含　　义
int setjmp(jmp_buf env);	长跳转时保存跳转环境信息
void longjmp(jmp_buf env, int _val);	恢复长跳转时的环境信息

库文件名：stdarg.h 或 cstdarg.hpp
用途：参变量列表函数，如表 3-9 所示。

表 3-9　参变量列表函数

函　数　原　型	含　　义
type va_arg(_ap, type);	向可变长参数列表中加入下一个类型参数
void va_end(_ap);	可变长参数列表结束
void va_start(_ap, parmN);	初始化一个可变长参数列表，并赋给它第一个参数类型

库文件名：string.h 或 cstring.hpp
作用：字符串函数，如表 3-10 所示。

表 3-10　字符串函数列表

函　数　原　型	含　　义
void *memchr(const void *cs, int c, size_t n);	在一个长序为 n 的字符串 cs 中查找字符 c
int memcmp(const void *cs, const void *ct,　size_t n);	比较字符串 cs 和 ct 的前 n 个字符
void *memcpy(void *s1, const void *s2, size_t n);	从字符串 s1 中拷贝 n 个字符到字符串 s2 中
void *memmove(void *s1, const void *s2, size_t n);	从字符串 s1 中转移 n 个字符到字符串 s2 中
void *memset(void *mem, int ch,size_t n);	将 ch 的值拷贝到 mem 指针指向的第一个字符串的长度值中
char *strcat(char *string1, const char *string2);	连接 s2 字符串到 s1 字符串的尾部
char *strchr(const char *string, int c);	查找字符串 string 中位置为 c 的字符
int strcmp(const char *s1, const char *s2);	比较字符串 s1 和字符串 s2，若 s1<s2，返回负数；若 s1= =s2，返回 0 值；若 s1>s2，返回正数
int strcoll(const char *string1, const char *string2);	比较字符串 string1 和字符串 string2，若 string1<string2，返回负数；若 string1= =string2，返回 0 值；若 string1>string2，返回正数
char *strcpy(char *dest, const char *src);	将字符串 src 拷贝到字符串变量 dest 中
size_t strcspn(const char *string,const char *chs);	返回所有不在字符串 chs 中但在字符串 string 中的字符所占的长度
char *strerror(int errno);	按错误号返回出错信息

续表

函　数　原　型	含　　义
size_t strlen(const char *string);	返回字符串的长度
char *strncat(char *dest, const char *src, size_t n);	从字符 src 中向字符串变量中加入 n 个字符
int strncmp(const char *string1, const char *string2, size_t n);	比较两个字符串常量中的前 n 个字符
char *strncpy(char *dest, const char *src, size_t n);	拷贝字符串常量 src 中的 n 个字符到字符串变量 dest 中
char *strpbrk(const char *string, const char *chs);	定位到字符串常量 string 中第一个属于字符串常量 chs 中的字符
char *strrchr(const char *string, int c);	查找字符串常量中最后出现的字符 c
size_t strspn(const char *string,const char *chs);	返回字符串 string 中由字符串常量 chs 中的字符组成的部分的长度
char *strstr(const char *string1, const char *string2);	发现字符串 string1 中出现字符串 string2 的位置
char *strtok(char *str1, const char *str2);	用字符串常量 str2 中的字母分隔字符串 str1
size_t　strxfrm(char *to, const char *from, size_t n);	将 n 个字符由字符串常量 from 传给字符串变量 to

库文件名：time.h 或 ctime.hpp

作用：时间函数，如表 3-11 所示。

表 3-11　时间函数列表

函　数　原　型	含　　义
char *asctime(const struct tm *timeptr);	将时间常量转换为字符串
clock_t clock(void);	返回程序执行时间
char *ctime(const time_t *timer);	将日期转换为字符串形式的本地时间
double difftime(time_t time1, time_t time0);	返回两个日期之间的时间间隔
struct tm *gmtime(const time_t *timer);	将本地时间转换为格林威治时间
struct tm *localtime(const time_t *timer);	将日期转换为本地时间
time_t mktime(struct tm *timeptr);	将本地时间转换为日期（或标准时间）
size_t strftime(char *s, size_t maxsize,const char *format, const struct tm *timeptr);	将时间转化为字符形式
time_t time(time_t *timer);	返回当前以秒的形式表示的日期

注：本地时间即为计算机的时钟时间。

3.6　CCStudio 库函数

如果 CCStudio 安装在默认目录下，即安装在 c:\ti 目录下，进入这个目录，可以看到几个子目录，其中 C5400 目录是专用于 C5400 系列 DSP 芯片的，C5500 目录是专用于 C5500 系列 DSP 芯片的，C6000 目录是专用于 C6000 系列 DSP 芯片的，其他的目录是 CCStudio 的集成环境文件，是被所有 TMS320 系列共用的。

C5400 目录和 C5500 目录下的子目录结构相似，其中的 cgtools 目录是 C/C++的通用支撑库和编译库，是上一节所介绍的部分；其他的如 BIOS 和 RTDX 目录将在后面章节中介绍，其中的 DSPLIB 子目录即为 CCStudio 提供的库函数及编译库等的文件所在的目录。这些库函数都是用汇编语言书写的，可以供 C/C++语言调用。在 dsplib.h 头文件中给出了调用这些函数的原型。在 54x_src 子目录下给出了这些程序的源代码，用户可以根据自己的需要修改。建议修改前先备份一份，然后将修改后的文件重新编译成一个支撑库，可以重新命名这个库为自己专用。在 examples 子目录下给出了所有这些函数的 C/C++测试程序，用户可以使用这些程序对这些库函数进行测试，当然全部库函数都是经过严格测试过的。

调用这些库函数有很多好处：一方面，这些库函数是用汇编语言所编写的，所以编译效率高，执行速度快；另一方面，这些库函数几乎包括了当前已经成熟的数字信号处理的大部分算法，用户直接调用可以减少不必要的劳动，可加快开发项目的软件设计速度。

C5400 和 C5500 都提供了通用的 DSPLIB 库，即通用数字信号处理算法库，C5500 还提供了 IMGLIB 库，即图像/视频信号处理算法库。下面仅介绍一下 C5400 的 DSPLIB 库的库函数及其用法和 C5500 的 IMGLIB 库的基本情况。

3.6.1　DSPLIB 库

在 c:\ti\c5400\dsplib\include\dsplib.h 中列出所有 DSPLIB 库函数的原型，在程序中调用这些库函数时必须包括该头文件，即在主程序中加入下面的代码：

```
#include "dsplib.h"
```

并将 54xdsp.lib 或 54xdspf.lib 加入到工程文件中，54xdspf.lib 针对超过 64 KW（64 K×16 bit）的大编译模式。对于 C5500 系列，对应的库文件名为：55xdsp.lib 或 55xdspx.lib。

在 dsplib.h 中给出以下方面的一些函数原型：

(1) 实数或是复数的 FFT 算法；
(2) 数字滤波和卷积；
(3) 自适应滤波；
(4) 相关；
(5) 数学函数；
(6) 三角函数；
(7) 矩阵运算；
(8) 杂项如数据格式转换等。

在 TI 的技术文档 spra480b.pdf 和 spru422.pdf 中分别详细地介绍了 C5400 和 C5500 的 DSPLIB 库函数的具体用法。下面给出用法示例，这个示例是对第 3.3 节所述程序的一点改进，即加入了一个 FIR 数字滤波函数，将 ADC 输出的数据进行滤波后再送到 DAC 去，这段程序插入点在上面的程序中有说明。除了这一点修改之外，还应在主文件头部加入：

#include “dsplib.h”

和几个变量定义，并把 54xdsp.lib 运行支撑库加入到 user_audio.pjt 工程文件中去，仅是主程序文件需要改动。下面给出这个改动后的主程序文件（硬件要求同前，用计算机向 SY-5402EVM 板提供音频信号，用耳机从 SY-5402EVM 接出来即可，并需要使用 MATLAB 生成 FIR 滤波器的系数）。

```
/*****************************************************************************/
/* filename: user_aduio.c                                                    */
/* Author:ZhangYong                                                          */
/* 2002-12                                                                   */
/*****************************************************************************/
    #include "user_type.h"
    #include "user_face.h"
    #include "user_func.h"
    #include "dsplib.h"                    //包括 dsplib.h 函数库头文件

    s16 in_data;                /*temp data*/
    s16 out_data;
    short db[64];
    short   *dbptr = &db[0];
    short sh_in[1],sh_r[1];
    short sh_h[64] ={
          13,    -32,    -31,     22,     52,    -16,    -84,     -9,    117,
          55,   -142,   -128,    147,    225,   -115,   -339,     30,    455,
         123,   -551,   -359,    597,    691,   -549,  -1143,    341,   1775,
         176,  -2825,  -1578,   5900,  13543,  13543,   5900,  -1578,  -2825,
         176,   1775,    341,  -1143,   -549,    691,    597,   -359,   -551,
         123,    455,     30,   -339,   -115,    225,    147,   -128,   -142,
          55,    117,     -9,    -84,    -16,     52,     22,    -31,    -32,
          13
    };                  //由 MATLAB 产生的滤波器系数
    // Main program
    void main()
    {
        unsigned int i=0;
```

```
    unsigned int j;

    for (j=0;j<64;j++)
        db[j]=0;
    sh_r[0]=0;

    init_board();
    while (1)
    {
            *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR;
            while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
            in_data=*(volatile u16*)DRR1_ADDR(1);
            in_data &= 0xFFFE;

            sh_in[0]=in_data;
            fir(sh_in, sh_h, sh_r, &dbptr, 64, 1);     //加入的 FIR 滤波函数
            out_data=sh_r[0];
            out_data=out_data & 0xFFFE;

            *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;
            while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
            *(volatile u16 *)DXR1_ADDR(1)=out_data;

            i++;
            if(i>1024)
            {
                *(volatile u16 *)reg_ST1 ^= 0x2000;     //Every 1024 times XF changed
                i=0;
            }
    }
}
```

加入一个低通滤波器之后，听音乐的效果要差一些。

3.6.2　IMGLIB 库

IMGLIB 库是 C5500 系列独有的一个插件库，其中的库函数也是用优化的汇编语言编写的，可以供 C/C++语言调用。该库主要分为三部分，分别完成图像压缩、图像分析、图像滤波和格式转化等功能。值得一提的是这个库里面集成了小波的功能。调用这个库时，

在头文件中应包括 imglib.h 或 wavelet.h，并将 55ximg.lib 或 55ximgx.lib 加入到工程文件中去。这个库中的每个函数也提供了源程序和示例，这个库目前不是免费的。

3.7　C++ 类

类也是一种数据类型，它本质的特性是数据封装、继承和多态。类定义的变量称为对象（抽象类不能定义变量，只能作为基类，本书没有提及）。只有 C++才支持类。

3.7.1　类的概念

类是一种可以定义数据结构和建立在这个数据结构基础上的方法的集总数据类型，使用关键字 class 来定义。如：

```
class cl_audio        //定义一个类，类名为 cl_audio
{
    private:   //类可以有私有成员（成员包括变量和函数）
        short sh_aud_in;    //下面定义了两个私有成员数据，只有类的成员函数才能访
        short sh_aud_out;   //问，友元函数也可以访问
    public:    //类可以有公有成员
        cl_audio(short vr_audin,short vr_audout);    //构造函数用于对类的私有数据
        void   vd_rd_bsp1();                         //成员初始化
        void   vd_wt_bsp1();
        ~cl_audio();                    //析构函数，类结束时自动调用
        virtual void   vd_agc();        //虚函数可以被子类同名同参的函数重载
    protected:      //类可以有保护成员
        void   vd_pcm();                //保护成员函数可以被子类访问
    friend   void   fr_rw_bsp1();       //类的友元函数可以访问类的一切成员数据
};
```

类中定义成员的访问方法可以有以下几种类型：私有的，公有的，保护的。如果没有明确指出，则默认为私有的。每一种访问类型都可以包括数据成员和函数成员（也称为方法或功能）；私有成员和保护成员类外是不可见的（即不可访问的）；私有成员（private）只能被同类的成员函数或是友元函数调用；保护成员（protected）可以被同类的成员函数、友元函数或是派生的子类所访问；而公有成员（public）对外是透明的。

函数重载有两种方法：一种是前面讲过的参数不同函数名相同的重载方法；另一种就是这里讲述的同名同参的类与子类之间的虚函数的重载方法。类的同名函数称为构造函数，可以在定义对象的同时对它初始化，构造函数可以重载。析构函数是在类名前加一个波浪号，它一般不被显示、定义，主要作用是对象释放时会自动调用析构函数释放掉所有占用的资源。类中定义成员变量和成员方法的方法与类外的定义方法相同。友元函数不属于类的成员，它可以访问类的一切成员，虽然破坏了类的数据封装特性，但它是可以继承的。在声明类的成员函数的时候，形参可以按从右向左的顺序进行初始化，即最右边的形参初

始化之后，才能对它左边的一个形参初始化。在声明类的成员方法的时候也可以把函数体直接写在下面，但一般不这样做，为了使类看上去简洁。类的定义必须以“;”结尾。

类的继承是代码复用和进行功能扩展的一种方法，是在原有的类的基础上派生出新的类并增强其中的某些方法。其语法如下：

定义一个子类，子类名为 cl_audio_agc。

```
class cl_audio_agc:public cl_audio        //对基类的 public 进行继承
{
    private:
        short    sh_over;
    public:
        void vd_agc();              //重载了基类的虚函数
    protected:
};
```

类可以实现多重继承，子类也可以再派生出新的子类。如：

```
class cl_agc2:public cl_audio_agc;
```

在多继承时，可以使用虚基类，如：

```
class cl_A;
class cl_X:virtual public A;
class cl_Y:virtual public B;
class cl_Z:public X,public Y
```

对类中成员方法的定义有两种：一种是可以将函数体写到类中；另外一种是将函数体写到类外的方法，如下例：

```
void cl_audio:: vd_rd_bsp1()
{
    函数体;
}
```

类定义对象的方法如下：

```
cl_audio    ob_audio;   //ob_audio 为一个对象
```

使用对象可以访问它的公有函数，示例如下：

```
ob_audio.vd_rd_bsp1();
```

还可以定义对象指针，对象指针对成员的访问使用运算符“->”。在类中还可以有其他类的对象或公有方法。

可见，类这种数据类型是将成员数据和成员方法封装在类中，这些方法的实现对用户是一个黑箱，而仅告诉用户操作这些方法的界面，用户不用关心其具体的运作过程，只需要使用即可。即内容实现过程是不可见的，操作方法是透明的。通过这种方法，类达到了对数据的封装效果。唯一可以破坏这个黑箱的函数就是这个类的友元函数。

短短的一页不可能讲清类这种数据类型，读者可以在实践中进一步学习，也可以参考

相关的资料。下一节我们将给出一个完整的实例，以加深对它的理解。

3.7.2　程序实例

这个实例使用类的方法来完成第 3.3 节介绍的 C 语言程序的功能。硬件准备同前。使用计算机向 SY-5402EVM 送出语音信号，可以用耳机听到清晰的声音。每个 C/C++源文件后面至少要有一个空行，CCStudio 要求这样。

程序需要 rts.lib 的支持，需要加入 vectors.asm 和 user_audio.cmd 文件，工程中其他的原文件 user_class.h、user_func.cpp 和 cadd_audio.cpp 清单列出如下，同时给出了 MAP 表便于读者比较。本章中两次列出 MAP 表的详细内容，是想表明 MAP 表在开发设计中具有相当重要的地位。

本 C++程序实例实现了和第 3.3 节介绍的 C 程序完全相同的功能，而且，C++中去掉了原 C 程序中辅助性的几句代码(这几句代码是用于观察 DSP 存储空间的)。下表 3-12 是 C++和 C 编译成的可执行文件的对比，由此可以清楚地看出为什么不提倡使用 C++语言了。

表 3-12　C++和 C 的可执行文件的大小比较

项　　目	C++的可执文件	C 的可执行文件
Debug（调试版）	9.38 KB	4.42 KB
Release（发行版）	6.26 KB	2.70 KB

图 3-11 是使用 C++语言程序的编译环境，供读者参考。

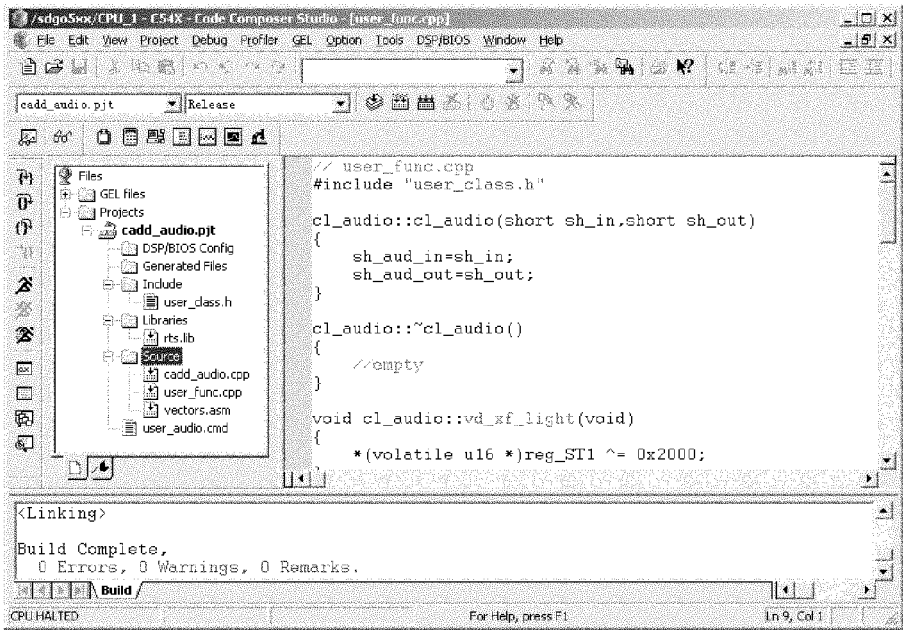


图 3-11　C++语言的编译工程及环境

下面是程序源代码：

文件 user_class.h 的内容如下：

```
// user_class.h
/* Register Definition    MCBSP    */

#define SPSA_ADDR(port)          (port ? 0x48 : 0x38)
#define SPSD_ADDR(port)          (port ? 0x49 : 0x39)

#define DRR2_ADDR(port)          (port ? 0x40 : 0x20)
#define DRR1_ADDR(port)          (port ? 0x41 : 0x21)
#define DXR2_ADDR(port)          (port ? 0x42 : 0x22)
#define DXR1_ADDR(port)          (port ? 0x43 : 0x23)

#define MCBSP_ACCSUB_ADDR(port) (port ? 0x49 : 0x39)

#define    SPCR1_SUBADDR     0x00
#define    SPCR2_SUBADDR     0x01
#define    RCR1_SUBADDR      0x02
#define    RCR2_SUBADDR      0x03
#define    XCR1_SUBADDR      0x04
#define    XCR2_SUBADDR      0x05
#define    SRGR1_SUBADDR     0x06
#define    SRGR2_SUBADDR     0x07
#define    MCR1_SUBADDR      0x08
#define    MCR2_SUBADDR      0x09
#define    RCERA_SUBADDR     0x0A
#define    RCERB_SUBADDR     0x0B
#define    XCERA_SUBADDR     0x0C
#define    XCERB_SUBADDR     0x0D
#define    PCR_SUBADDR       0x0E

//SPCR10        39h   SPCR11    49h   00h   Serial port control register 1
#define         bsp_SPCR11          0x0021
//SPCR20        39h   SPCR21    49h   01h   Serial port control register 2
#define         bsp_SPCR21          0x0201
//PCR0          39h   PCR1      49h   0Eh   Pin control register
#define         bsp_PCR1            0x000C
//RCR10         39h   RCR11     49h   02h   Receive control register 1
```

```
#define         bsp_RCR11           0x0040
//RCR20         39h  RCR21    49h  03h  Receive control register 2
#define         bsp_RCR21           0x0000
//XCR10         39h  XCR11    49h  04h  Transmit control register 1
#define         bsp_XCR11           0x0040
//XCR20         39h  XCR21    49h  05h  Transmit control register 2
#define         bsp_XCR21           0x0000
//SRGR10        39h  SRGR11   49h  06h  Sample rate generator register 1
#define         bsp_SRGR11          0x0000
//SRGR20        39h  SRGR21   49h  07h  Sample rate generator register 2
#define         bsp_SRGR21          0x0000
//MCR10         39h  MCR11    49h  08h  Multichannel register 1
#define         bsp_MCR11           0x0000
//MCR20         39h  MCR21    49h  09h  Multichannel register 2
#define         bsp_MCR21           0x0000
//RCERA0        39h  RCERA1   49h  0Ah  Receive channel enable register partition A
//#define       RCERA1
//RCERB0        39h  RCERB1   49h  0Bh  Receive channel enable register partition B
//#define       RCERB1
//XCERA0        39h  XCERA1   49h  0Ch  Transmit channel enable register partition A
//#define       XCERA1
//XCERB0        39h  XCERB1   49h  0Dh  Transmit channel enable register partition B
//#define       XCERB1

//---------CPU-----------
//ST1       addr:0x0007     Status register 1   ST1[13]=XF
#define       reg_ST1             0x0007
#define       ST0                 *(volatile unsigned int*)0x06
#define       ST0_ADDR            0x06

#define     PMST            0x001D
#define     SWWSR           0x0028
#define     SWCR            0x002B
#define     BSCR            0x0029
#define     CLKMD           0x0058

#define       PMST_VAL          0x00A0   //interupt vectors from 0x80
#define       SWWSR_VAL         0x8fff
#define       SWCR_VAL          0x0001
```

```
#define      BSCR_VAL        0x8802
#define      CLKMD_VAL       0x9FF7

typedef unsigned int u16;
class cl_audio                              //定义了一个类 cl_audio
{
    private:                                //类的私有成员
        short   sh_aud_in;
        short   sh_aud_out;
    public:                                 //类的公有成员
        cl_audio(short ,short);             //类的构造函数
        void    init_board(void);           //初始化函数
        void    vd_rd_bsp1(void);           //读串口 1
        void    vd_agc(void);               //自动增益控制算法，在此为空
        void    vd_wt_bsp1(void);           //写串口 1
        void    vd_xf_light(void);          //XF 闪烁
        ~cl_audio();                        //or ~cl_audio(){} -without define later   析构函数

};   //类的结尾必须有“;”号
```

文件 user_func.cpp 的内容如下：

```
// user_func.cpp
#include "user_class.h"

cl_audio::cl_audio(short sh_in,short sh_out)        //在类外定义类中的函数
{
    sh_aud_in=sh_in;
    sh_aud_out=sh_out;
}

cl_audio::~cl_audio()                               //析构函数一般为空
{
    //empty
}

void cl_audio::vd_xf_light(void)
{
    *(volatile u16 *)reg_ST1 ^= 0x2000;
}
```

```
void cl_audio::init_board(void)                    //初始化 SY-5402EVM 板
{
    *(volatile u16 *)CLKMD = 0x0000;
    while(*(volatile u16 *)CLKMD & 0x0001){};
    *(volatile u16 *)CLKMD = CLKMD_VAL;

    *(volatile u16 *)PMST = PMST_VAL;
    *(volatile u16 *)SWWSR =SWWSR_VAL;
    *(volatile u16 *)SWCR =SWCR_VAL;
    *(volatile u16 *)BSCR =BSCR_VAL;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //select SPCR11
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SPCR11;
    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SPCR21;
    *(volatile u16 *)SPSA_ADDR(1)=RCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_RCR11;
    *(volatile u16 *)SPSA_ADDR(1)=RCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_RCR21;
    *(volatile u16 *)SPSA_ADDR(1)=XCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_XCR11;
    *(volatile u16 *)SPSA_ADDR(1)=XCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_XCR21;
    *(volatile u16 *)SPSA_ADDR(1)=SRGR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SRGR11;
    *(volatile u16 *)SPSA_ADDR(1)=SRGR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SRGR21;
    *(volatile u16 *)SPSA_ADDR(1)=MCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_MCR11;
    *(volatile u16 *)SPSA_ADDR(1)=MCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_MCR21;
    *(volatile u16 *)SPSA_ADDR(1)=PCR_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_PCR1;
}

void   cl_audio::vd_rd_bsp1(void)              //定义读串口 1 的函数体
{
    *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR; //Receive Data from McBSP1
```

```
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    sh_aud_in=*(volatile u16*)DRR1_ADDR(1);
    sh_aud_in=sh_aud_in & 0xFFFE;
}

void   cl_audio::vd_adm(void)
{
    sh_aud_out=sh_aud_in;
}

void   cl_audio::vd_wt_bsp1(void)              //定义写串口 1 的函数体
{
    sh_aud_out=sh_aud_out & 0xFFFE;
    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;   //Transmit Data To McBSP1
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=sh_aud_out;
}
```

文件 cadd_audio.cpp 的内容如下：

```
// filename: cadd_audio.cpp   主程序
// Author:ZhangYong
// 2002-12
// cadd_audio.pjt (cadd_audio.cpp,user_func.cpp,user_class.h,
// vectors.asm,user_audio.cmd)

#include "user_class.h"

// Main program
int main()                          //主程序入口地址
{
    cl_audio obj_audio(0,0);
    unsigned int i=0;
    obj_audio.init_board();
    while (1)
    {
        obj_audio.vd_rd_bsp1();//read from ADC
        obj_audio.vd_adm();       //code modulation
        obj_audio.vd_wt_bsp1();//write to DAC
```

```
            i++;
            if(i>1024)
            {
                obj_audio.vd_xf_light();    //XF light
                i=0;
            }
        }
}
```

文件 cadd_audio.map 的内容如下：

```
******************************************************************************
TMS320C54x COFF Linker                  PC Version 3.70
******************************************************************************
>> Linked Fri Dec 20 18:33:28 2002

OUTPUT FILE NAME:     <./Release/cadd_audio.out>
ENTRY POINT SYMBOL: "_c_int00"    address: 00000180
/*可以从上面看出本程序的入口地址为 0x180*/

MEMORY CONFIGURATION

                  name      origin          length          used            attr        fill
                  --------- --------        ---------       --------        ----        --------
PAGE    0:        VECS      00000080        00000100        00000064        RIX
                  PROG      00000180        00003000        000004b8        RWIX

PAGE    1:        DATA      00003080        00000f80        0000082a        RWI
/*上面为存储区域的配置*/

/*下面为各段的配置*/
SECTION ALLOCATION MAP

output                                                    attributes/
section     page        origin          length          input sections
--------    ----        ----------      ----------      ----------------
.text       0           00000180        000004a2
                        00000180        0000004a        rts.lib : boot.obj (.text)
                        000001ca        00000000        vectors.obj (.text)
```

```
                    000001ca    00000054    rts.lib : exit.obj (.text)
                    0000021e    00000007            :_lock.obj (.text)
                    00000225    0000002b    cadd_audio.obj (.text)
                    00000250    000000e8    user_func.obj (.text)
                    00000338    0000000a    rts.lib : del_sof.obj (.text)
                    00000342    00000275            : memory.obj (.text)
                    000005b7    00000025            : new_sof.obj (.text)
                    000005dc    0000002c            : memmov.obj (.text)
                    00000608    0000000b            : new_.obj (.text)
                    00000613    0000000f            : memcpy.obj (.text)

.cinit      0       00000622    00000016
                    00000622    00000009    rts.lib : exit.obj (.cinit)
                    0000062b    00000006            : _lock.obj (.cinit)
                    00000631    00000003            : memory.obj (.cinit)
                    00000634    00000003            : new_.obj (.cinit)
                    00000637    00000001    --HOLE-- [fill = 0000]

.pinit      0       00000180    00000000

.vectors    0       00000080    00000064
                    00000080    00000064    vectors.obj (.vectors)

.stack      1       00003080    00000400    UNINITIALIZED
                    00003080    00000000    rts.lib : boot.obj (.stack)

.bss        1       00003480    0000002a    UNINITIALIZED
                    00003480    00000000    rts.lib : boot.obj (.bss)
                    00003480    00000000            : memcpy.obj (.bss)
                    00003480    00000000            : memmov.obj (.bss)
                    00003480    00000000            : new_sof.obj (.bss)
                    00003480    00000000            : del_sof.obj (.bss)
                    00003480    00000000    vectors.obj (.bss)
                    00003480    00000000    user_func.obj (.bss)
                    00003480    00000000    cadd_audio.obj (.bss)
                    00003480    00000023    rts.lib : exit.obj (.bss)
                    000034a3    00000002            : _lock.obj (.bss)
                    000034a5    00000003            : memory.obj (.bss)
                    000034a8    00000002            : new_.obj (.bss)
```

```
.const      1         00003080    00000000    UNINITIALIZED

.switch     1         00003080    00000000    UNINITIALIZED

.sysmem     1         000034aa    00000400    UNINITIALIZED
                      000034aa    00000000    rts.lib : memory.obj (.sysmem)

.cio        1         00003080    00000000    UNINITIALIZED

.far        1         00003080    00000000    UNINITIALIZED

.data       1         00000000    00000000    UNINITIALIZED
                      00000000    00000000    rts.lib : boot.obj (.data)
                      00000000    00000000            : memcpy.obj (.data)
                      00000000    00000000            : new_.obj (.data)
                      00000000    00000000            : memmov.obj (.data)
                      00000000    00000000            : new_sof.obj (.data)
                      00000000    00000000            : memory.obj (.data)
                      00000000    00000000            : del_sof.obj (.data)
                      00000000    00000000    vectors.obj (.data)
                      00000000    00000000    user_func.obj (.data)
                      00000000    00000000    cadd_audio.obj (.data)
                      00000000    00000000    rts.lib : _lock.obj (.data)
                      00000000    00000000            : exit.obj (.data)
/*下面给出了全局的符号定义及地址，分了两种方式给出：一种是按 ASCII 顺序，另一种是按符号地址*/
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

Address         name
--------        ----
00003480        .bss
00000000        .data
00000180        .text
0000021b        C$$EXIT
00000400        _STACK_SIZE
00000400        _SYSMEM_SIZE
00003480        _bss_
00000622        _cinit_
```

```
00000250    _ct_8cl_audioFsT1
00000000    _data_
00000338    _dl_FPv
00000328    _dt_8cl_audioFv
00000000    _edata_
000034aa    _end_
00000622    _etext_
000005b7    _nw_FUi
ffffffff    _pinit_
00000180    _text_
000034a1    _cleanup_ptr
000034a2    _dtors_ptr
00000001    _lflags
000034a3    _lock
0000021e    _nop
0000021f    _register_lock
00000222    _register_unlock
000034aa    _sys_memory
000034a4    _unlock
0000021b    _abort
000001f9    _atexit
00000180    _c_int00                /*看一下_c_int00 的地址*/
0000043d    _calloc
0000060e    _default_new_handler__3stdFv
000001ca    _exit
00000525    _free
0000026f    _init_board_8cl_audioFv
00000225    _main                   /*这是主函数的地址*/
000003d4    _malloc
00000613    _memcpy
000005dc    _memmove
000003a2    _minit
000034a9    _new_handler_fun_3std
000034a8    _nothrow_3std
0000045f    _realloc
00000608    _set_new_handler_3stdFPFv_v
000002fe    _vd_adm_8cl_audioFv
000002e8    _vd_rd_bsp1_8cl_audioFv
00000307    _vd_wt_bsp1_8cl_audioFv
```

```
00000321     _vd_xf_light_8cl_audioFv
00000622     cinit
00000000     edata
000034aa     end
00000622     etext
ffffffff     pinit

/*下面给出了按符号占据地址的顺序排列*/
GLOBAL SYMBOLS: SORTED BY Symbol Address

address      name
--------     ----
00000000     edata
00000000     _edata_
00000000     _data_
00000000     .data
00000001     _lflags
00000180     _text_
00000180     .text
00000180     _c_int00
000001ca     _exit
000001f9     _atexit
0000021b     C$$EXIT
0000021b     _abort
0000021e     _nop
0000021f     _register_lock
00000222     _register_unlock
00000225     _main
00000250     _ct_8cl_audioFsT1
0000026f     _init_board_8cl_audioFv
000002e8     _vd_rd_bsp1_8cl_audioFv
000002fe     _vd_adm_8cl_audioFv
00000307     _vd_wt_bsp1_8cl_audioFv
00000321     _vd_xf_light_8cl_audioFv
00000328     _dt_8cl_audioFv
00000338     _dl_FPv
000003a2     _minit
000003d4     _malloc
00000400     _SYSMEM_SIZE
```

```
00000400    _STACK_SIZE
0000043d    _calloc
0000045f    _realloc
00000525    _free
000005b7    _nw_FUi
000005dc    _memmove
00000608    _set_new_handler_3stdFPFv_v
0000060e    _default_new_handler_3stdFv
00000613    _memcpy
00000622    _etext_
00000622    cinit
00000622    _cinit_
00000622    etext
00003480    .bss
00003480    _bss_
000034a1    _cleanup_ptr
000034a2    _dtors_ptr
000034a3    _lock
000034a4    _unlock
000034a8    _nothrow_3std
000034a9    _new_handler_fun_3std
000034aa    end
000034aa    _end_
000034aa    _sys_memory
ffffffff    pinit
ffffffff    _pinit_

[53 symbols]
/*比 C 语言的 MAP 表多出了很多全局符号*/
```

3.8　C/C++文件操作

面向 DSP 的 C/C++程序设计中，C/C++的文件操作包括对通用计算机上的标准外设的输入和输出操作，这在调试 DSP 应用系统或功能板时可能用得上。但是，这部分可以访问通用计算机的资源与 RTDX 技术有本质的区别。RTDX 技术是进行实时数据交换的通信编程技术，而这里对通用计算机来说的 C/C++的 I/O 操作是不能进行实时操作的。也就是说，这部分内容对于面向 DSP 的 C/C++程序设计是没有多大意义的（也有例外：可以添加 DSP 设备用于 DSP 的外设访问）。将这些函数列于表 3-13。这些函数是被 CCStudio 支持的，在

调试 DSP 功能板需要用到它们时，该表内容可以作为参考。

库文件名：stdio.h 或 cstdio.hpp

用途：通用计算机的 I/O 函数，如表 3-13 所示。

表 3-13　通用计算机的 I/O 函数

函　数　原　型	含　　义
int add_device(char *name, unsigned flags,int (*dopen)(), int (*dclose)(),int (*dread)(), int (*dwrite)(),fpos_t (*dlseek)(), int (*dunlink)(), int (*drename)());	向设备列表中添加设备记录
void clearerr(FILE *_fp);	清除文件指针结束或指针错误
int fclose(FILE *_fp);	关闭文件
int feof(FILE *_fp);	检测文件指针是否指到文件尾
int ferror(FILE *_fp);	检测文件指针错误
int fflush(register FILE *_fp);	文件指针缓冲区是否满载
int fgetc(register FILE *_fp);	从文件中取得下一个字符
int fgetpos(FILE *_fp, fpos_t *pos);	将 pos 指向的内容在文件指针 fp 指向的文件中的当前位置记录下来
char *fgets(char *_ptr, register int _size, register FILE *_fp);	从文件指针 fp 指向的文件中读取长度为_size-1 的字符串，存入起始地址为_prt 的字符串中
FILE *fopen(const char *_fname,const char *_mode);	以_mode 指定的方式打开文件名为_fname 的文件
int fprintf(FILE *_fp, const char *_format, ...);	向文件中写入格式化数据
int fputc(int _c, register FILE *_fp);	向文件中写入一个字符
int fputs(const char *_ptr, register FILE *_fp);	向文件中写入一个字符串
size_t fread(void *_ptr, size_t _size, size_t _count, FILE *_fp);	从文件中读取长度为_size 的_count 个数据项写入_ptr 指向的地址空间
FILE *freopen(const char *_fname,const char *_mode, register FILE *_fp);	使用模式_mode 打开文件_fname，使文件指针指向该文件
int fscanf(FILE *_fp, const char *_fmt, ...);	从文件中读取格式化数据
int fseek(register FILE *_fp, long _offset,int _ptrname);	将文件指针从地址_ptrname 出发偏移_offset
int fsetpos(FILE *_fp, const fpos_t *_pos);	将文件指针定位在_pos 处
long ftell(FILE *_fp);	返回当前文件指针的位置

续表

函　数　原　型	含　　义
size_t fwrite(const void *_ptr, size_t _size,size_t _count, FILE *_fp);	将_ptr 指向的长度为_size 的_count 个地址内容写到文件中
int getc(FILE *_fp);	从文件中读取下一个字符
int getchar(void);	从标准输入设备输入一个字符
char *gets(char *_ptr);	从标准输入设备输入字符串
void perror(const char *_s);	按错误号显示出错信息
int printf(const char *_format, ...);	输出格式化数据到标准输出设备
int putc(int _x, FILE *_fp);	向文件中写入一个字符
int putchar(int _x);	向标准输出设备输出一个字符
int puts(const char *_ptr);	向标准输出设备输出一个字符串
int remove(const char *_file);	删除文件名
int rename(const char *_old_name, const char *_new_name);	更换文件名
void rewind(register FILE *_fp);	将文件指针复位到文件头
int scanf(const char *_fmt, ...);	从标准输入设备输入格式化数据
void setbuf(register FILE *_fp, char *_buf);	将一个地址缓冲区与一个文件缓冲区相
int setvbuf(register FILE *_fp, register char *_buf, register int _type, register size_t _size);	关联一个地址缓冲区与一个文件缓冲区
int sprintf(char *_string, const char *_format, ...);	向标准输出设备或文件输出格式化数据
int sscanf (const char *_str,　const char *_fmt, ...);	从标准输入设备或文件输入格式化数据
FILE *tmpfile(void);	创建临时文件
char *tmpnam(char *_s);	生成一个有效的文件名字符串
int ungetc(int _c, FILE *_fp);	向文件写入字符
int vfprintf(FILE *_fp, const char *_format, char * _ap);	向文件写入格式化数据
int vprintf(const char *_format, char * _ap);	向输出设备输出格式化数据
int vsprintf(char *_string, const char *_format, char *_ap);	向输出设备输出格式化数据

3.9 本 章 小 结

通过本章的学习可以使读者真正了解使用 C/C++语言开发 DSP 应用系统的基本方法和步骤，要充分掌握本章的内容需要有 EVM 板的工作环境。若没有这个条件，可以按第一章的介绍使 CCStudio 运行在模拟环境（Simulator）下，这时大部分程序需要修改一下。

本章开始给出的那个完整的程序，是在三意电子的 SY-5402EVM 板上成功运行过的，这个程序已经包含了面向 DSP 的 C/C++语言的基本特征。在第四章中，我们将用 DSP/BIOS 程序设计方法把这个程序重新写了一个版本。在第七章讲述 Boot Loader 时，通过计算机借助仿真器在线下载到 SY-5402EVM 板上的程序也是本章的这个程序（user_audio.out）。因为没有具体的语音信号处理算法，所以没有必要使用“裁缝师”，在混合语言模式下，一眼即可看出访问端口所用的代码只有七行左右。

从本章第 3.4 小节开始主要讲述 C/C++语言的语句、数据结构和库函数，这些内容是 C/C++程序设计人员必备的知识。值得一提的是，所有这些语句、数据结构和库函数作者均在 SY-5402EVM 板上作了测试，验证了其合法性和正确性。因为 CCStudio 中 C5400 和 C5500 有其独立的目录，这些目录包括支持文件和库，大部分内容的应用具有类似性。

如果读者具有 CCStudio 平台，经过本章学习后，应具备了 C/C++编写 DSP 程序的理论知识和动手能力。下面对本章开始时提出的思考题做一简要的回答。

DSPLIB 库和 IMGLIB 库是 TI 公司专项开发设计的程序包。这些程序包按照 DSP 算法标准设计，全部采用汇编语言编写，供 C 语言调用，执行效率高、速度快是这些库函数的最大特点。对于 C5000 定点 DSP 来说，这些库函数采用了统一的数据结构，即库函数的数据结构均为 Q15 或 Q31 等。

C 语言访问 VC5402 片上外设唯一的方法就是借助于指针，访问这些片上外设映射到映射存储器中的地址，从而达到访问片上外设的目的。需要说明的是，对于这些外设的访问，一般必须加上关键字 volitale，否则 C/C++程序在优化时会将这些标志符优化掉，就达不到访问的目的。在计算机模拟环境下，也可以访问这些“外设”，但遗憾的是，从 Watch 窗口却看不到数值。据说，最新的 CCStudio 2.1 中可以看到这些数值。

DSP 的混合语言编程曾经被认为是 DSP 程序设计发展的主流，其实不然。DSP 的混合语言编程只是在 C/C++语句无法完成的地方才插入几个汇编语句，如果 C/C++本身可以完成，而故意采用混合语言进行程序设计，其结果可能不在于提高速度，反而会使程序的完整性有损。更重要的是，随着 CCStduio 编译技术的不断提高，C/C++语言的优化效率也越来越高，C/C++语言硬件编程将得到更广泛的应用。混合语言编程有两种含义：其一是在 C/C++程序中调用汇编语句，使用 asm(“汇编语句”)；其二是在汇编程序中使用 C 语言函数，这常常通过标号跳转的方法来实现。

关于如何优化 C 语言程序的论题，涉及很多方面的内容。C 语言的优化一方面与编程者的程序设计水平有关，另一方面与编程者对 C/C++语言掌握的程度有关。只要程序设计人员能灵活地运用“裁缝师”，都可以找到程序中最不合理的部分，然后，将时间集中在这

些程序片段上进行修改，最后都有望设计出最优化的 C/C++程序来。此外，编译器提供了四级优化，打开 Project/Build Options/Compiler/Basic/Opt Level，有五种选择。作者在本章程序中均选用了-o2 优化方式，读者可以根据自己的需要选用，详细的含义请参考 CCStudio 的“帮助”文件。

关于 C5000 定点 DSP 支持浮点 C/C++数据类型和编程的问题，回答显然是肯定的。但是，没有必要为此持乐观态度。因为这个支持是用速度换来的，在要求实时性很强的环境下，建议应尽可能地减少不必要的浮点运算。

习　题　三

1. 面向 DSP 的 C/C++程序设计的原则是什么？
2. 面向 DSP 的 C/C++程序的设计流程是什么样的？
3. 面向 DSP 的 C/C++程序的设计框架是什么样的？
4. 如何编程将 TMS320VC5402 的时钟频率由 10 MHz 调整为 100 MHz？
5. 如何定义 DSB 的串行口寄存器地址？
6. 如何从 DSP 的串行口读取数据？
7. 如何向 DSP 的串行口写入数据？
8. 如何初始化 TMS320VC5402 的 CPU？
9. DSP 的初始化包括哪些工作？
10. 如何用 C/C++语言访问 DSP 的 I/O 空间？
11. 在中断向量表中如何调用中断？C 语言与 C++语言有什么不同？
12. 编写一个简单的程序，实现从串口读入数据，并将读入的数据送到另一个串口输出。
13. 在存储器配置文件中的 R、W、X 和 I 分别代表什么含义？
14. 如何从 MAP 表中找到程序的入口地址？
15. MAP 表的作用是什么？
16. C++程序生成的 MAP 表与 C 程序生成的 MAP 表有何不同之处？
17. C/C++有哪些基本的数据类型？
18. 如何定义 C/C++的常量及变量？
19. C/C++程序中为什么一般必须有“死循环”？
20. 在 C/C++程序中如何设计中断服务程序？
21. C/C++的运算符有哪些？
22. 简述 C/C++语言中的控制语句的使用方法。
23. C/C++语言中如何自定义一个函数？
24. C++函数重载有哪两种方法？
25. DSPLIB 库有什么特点？为什么说应尽可能地使用这个库里的函数而不是使用通用的 C/C++库函数？
26. 编写一个使用 DSPLIB 库中的低通滤波器库函数的小程序。

27. IMGLIB 库有哪些功能？
28. 如何定义一个 C++类？
29. C++的封装、继承和多态的含义是什么？
30. 如何定义 C++类中的函数体？
31. C++类如何初始化？
32. C/C++的文件操作在面向 DSP 的程序设计中主要应用在哪些方面？
33. 如何使用 C/C++的文件操作函数访问 DSP 片内资源？
34. C 语言程序设计有没有可能完全替代汇编语言？

第四章

DSP/BOIS 程序设计

4.1 本章内容简介

本章将系统地介绍 DSP/BIOS 编程的方法。同样地，本章也首先以一个完整的 DSP/BIOS 程序为例来介绍。在详细介绍这个程序的基础上，分为两部分分别详细地介绍 DSP/BIOS 组件及其中的插件 CSL，使读者充分掌握 DSP/BIOS 编程知识。

DSP/BIOS 开发环境为用户提供了一个图形化的用户界面，通过这个图形化的用户接口界面用户可以直观地访问 DSP 片上的各种资源，并可以对这些资源进行配置和初始化。此外，DSP/BIOS 还提供了代码生成器，用户在保存配置文件时，DSP/BIOS 为用户生成了所有需要的库函数文件。访问片上资源时主要是通过调用 API 函数来完成的。通过调用这些函数，用户可以方便地访问各种片上资源和外设。在调试方面，DSP/BIOS 支持多线程技术，使得在线调试分析成为可能。所谓在线调试就是在不影响 DSP 正常运行的情况下，即在不干扰 DSP 的各个工作和任务的情况下，实时地进行调试。这种在线调试具有更大的优越性，而传统的调试方法则必须终止 DSP 的运行才可以工作。在功能上，DSP/BIOS 是 TI 公司推出的完全免费的插件，具有多达约 200 个 API 函数可供调用，几乎涵盖了 DSP 上的一切片内资源的访问方法。

由于 DSP/BIOS 的优化能力很强，它的代码执行效率高；又由于 DSP/BIOS 为编程人员提供了一个完整的访问 DSP 片上硬件资源的程序框架，可为软件编程人员节约许多软件开发时间；且现在 DSP/BIOS 已迅速发展到了 2.0 版本，其强大而完备的功能，更具有吸引能力，因此掌握 DSP/BIOS 编程方法对软件设计人员来说是很有必要的。

思考题

(1) DSP/BIOS 的主要用途是什么？

(2) 使用 CSL 的方法有哪两种？

4.2　DSP/BIOS 编程实例

4.2.1　准备工作

DSP/BIOS 编程方法提供了图形化界面、代码快速原型*功能以及完备的 API 函数库，这些都极大地方便了用户访问 DSP 的片上资源和应用系统的在线调试。为了便于读者尽快地了解 DSP/BIOS，在这一节里先用一个实例介绍 DSP/BIOS。在一个程序中一旦使用了 DSP/BIOS，则 DSP/BIOS 的所有 API 库函数就都包括到工程下的 include 目录里了。API 库函数是按 DSP/BIOS 的组件严格分类的，相应组件的 API 库函数的文件名都是以组件名开头、以.h 或.h54 来结尾的（对于 C5500，以.h55 取代.h54）。虽然有众多的 API 文件，但是我们只关心那些我们要用到的文件。例如，要对 McBSP1 进行访问，可以查找相关的 API 库文件，并找到这些操作的函数原型。如果读者对这些函数原型很了解，就可不用去关心这些文件了。当然有一些关键性的文件一定要弄清，这些都将在下面的实例中作具体介绍。

DSP/BIOS 提供了一些类似于硬件测量设备的软测量设备，巧妙地运用这些软测量设备（下文统称为“示波器”）并与“裁缝师”配合，就可以对一个程序的运行情况达到透明化的认识程度。

此外，DSP/BIOS 有一些关于数据类型、存储器配置、语句（或称句柄）等等的约定，这些约定只是为了便于读者见名知义，易于理解。这些约定只是形式上的变动，对于面向 DSP 的 C/C++程序设计来说，通过 DSP/BIOS 访问了 DSP 片上资源之后，C/C++的库函数或是 DSPLIB 库函数便会进一步去完成要求的数据处理的算法。所以可以把 DSP/BIOS 看作是一个软的硬件平台。

硬件上的准备工作与第三章相同，本章的实例也是工作在三意电子的 SY-5402EVM 板上。通过计算机的耳机输出端送语音信号到 SY-5402EVM 板的语音输入端，程序运行过程中，可以通过 SY-5402EVM 板上接出的耳机听到清晰的声音。

4.2.2　开发过程

开发一个 DSP/BIOS 程序与开发一般的程序具有大体相同的步骤，但也有一点不同之处。所谓 DSP/BIOS 程序设计，是指通过使用 DSP/BIOS 的组件、API 函数和“示波器”等功能来辅助完成 C/C++程序设计的一种程序设计方法。在 DSP/BIOS 程序设计中，不再显式地定义中断向量表文件（vectors.asm），中断向量定义在.cmd 文件中。对于片上外设的访问是借助 DSP/BIOS 中的 CSL（片上外设支持库）来完成的。调用这个支持库也可以不在 DSP/BIOS 环境下完成，这时，需要软件开发人员对这个库的各项内容比较了解，特别是对片上外设的配置、初始化等更要清楚，而且没有图形化的支持，所以借助 DSP/BIOS 组件管理器来调用 CSL 是首选方案。所以说软件开发过程是从这个 DSP/BIOS 组件管理器开始的。

注：代码快速原型技术，又称代码快速成型技术，是指进行高级语言程序设计时，由代码编辑器自动生成程序框架，或者帮助程序设计人员生成函数头部和函数框架的方法。CCStudio 的 DSP/BIOS 提供了代码快速原型技术，它可以自动为用户生成完整的程序框架。

具体的开发过程如下：

？第一步：进入 CCStudio 集成环境，新建一个工程文件，点击 Project/New。如图 4-1 所示，工程文件名为 bios_audio.pjt，在 project 一栏中输入 bios_audio 即可。

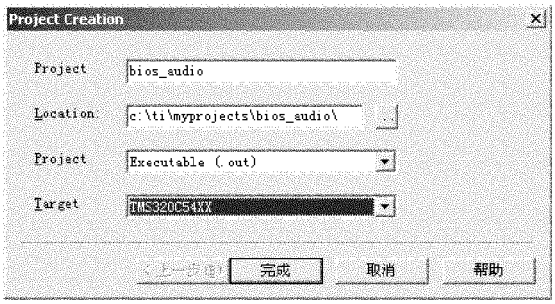


图 4-1　新建工程文件对话框

在图 4-1 中点击完成，进入图 4-2 所示的窗口，在这个窗口中可以看到工程文件管理器中只有一个空的工程文件 bios_audio.pjt。这是编写一切 DSP 程序需要做的第一步。

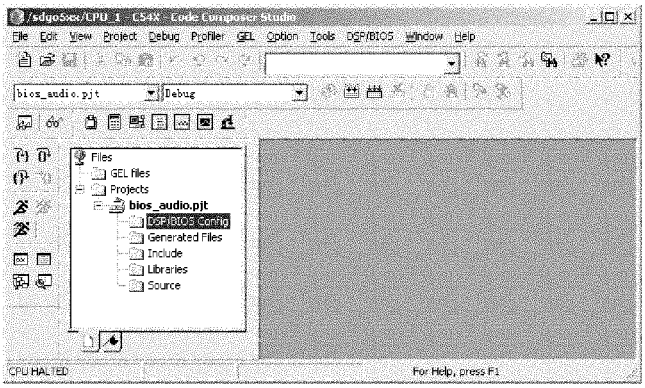


图 4-2　空的工程文件

？第二步：进入 DSP/BIOS 组件管理器中进行 DSP 片上资源的设置，并将生成的配置文件加入到工程文件中去。点击 File/New/DSP/BIOS Configuration 进入图 4-3 所示的对话框。

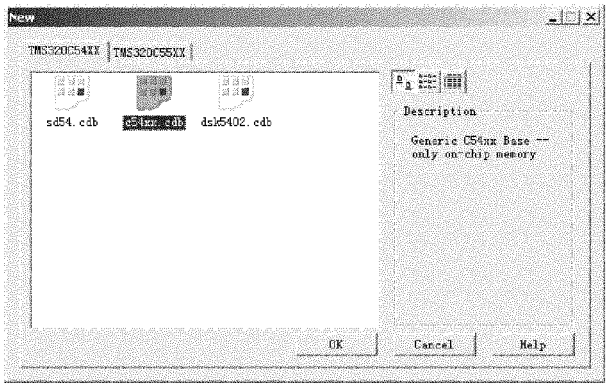


图 4-3　选择 DSP/BIOS 配置模板

在图 4-3 中，c54xx.cdb 为 C5400 系列通用的模板， dsk5402 是专用于 VC5402 的模板。为了使程序更具有示范意义，在这里选用了 c54xx.cdb 这一模板。点击“OK”进入 DSP/BIOS 组件管理器，将其保存为 bios_audio.cdb，如图 4-4 所示。

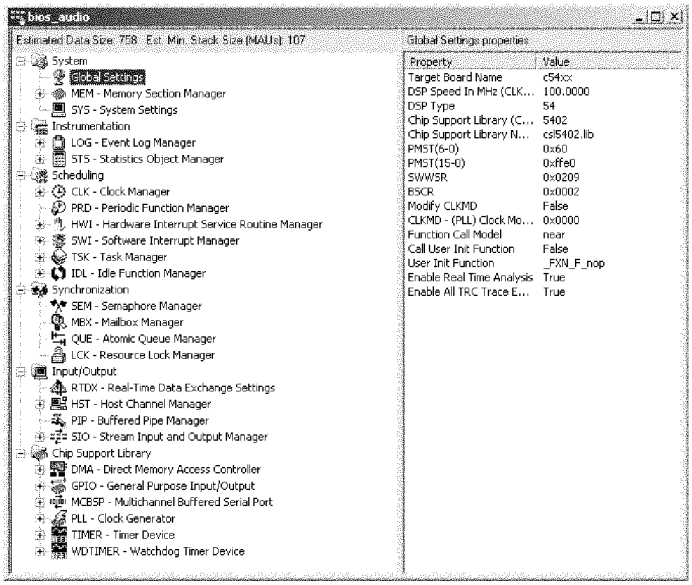


图 4-4　DSP/BIOS 组件管理器

在图 4-4 中包括了 VC5402（这里选用的是 VC5402）的所有片上资源，左边一栏为 DSP/BIOS 的组件区，按功能不同分为六部分。DSP/BIOS 的 API 函数分别支持各个部分中的不同模块。API 函数的函数名由图 4-4 中各模块的简称开头，然后加上一个下划线，之后的标识符代表该 API 函数的功能。如对 API 函数 CLK_getltime，它属于时钟管理器模块，所以以 CLK_开头，它完成的功能由 getltime 定义，即获得计时器的低分辨率格式。在 System 部分中 Global Settings 模块的缩写为 GBL，其他的缩写名在 DSP/BIOS 组件管理器中都明确给定了。此外，DSP/BIOS 还支持了一些小的汇编语言编写的 API 函数，称作原子功能函数或微功能函数，以 ATM 为这些函数的函数名头。关于这些模块的详细含义将在后面章节中介绍。

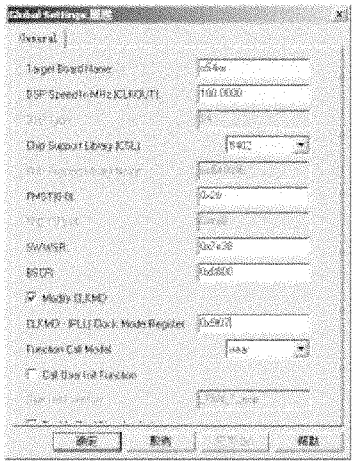


图 4-5　CPU 属性设置值

下面以 CPU 设置（GBL-Global Settings）、存储器配置（MEM-Memory Section Manager）、硬件中断设置（HWI-Hardware Interrupt Service Routine Manager）、多通道缓冲串行口设置（McBSP-Multichannel Buffered Serial Port）为例介绍 DSP/BIOS 图形化的操作方法。

(1) CPU 设置：在图 4-4 中选择 Global Settings，单击鼠标右键，在右键单击后的弹出菜单中选择 Properties，出现如图 4-5 所示的窗口。在目标板名中

输入 C54xx，DSP 的工作时钟一栏输入 100，片上支持库必须选择目标板上所用的 DSP 芯片，本例中选择 5402。下面给出几个 CPU 映射寄存器的设置，这可根据自己项目开发的需要进行设置，这里将其分别设置为：0x20，0x7e38，0x8800，0x9f07。函数调用模式必须与实际的函数调用情况相结合，在本应用实例中，没有使用超过 0xFFFF 的函数调用，所以选择 near 模式。

(2) 存储器配置：在图 4-4 中选择 MEM-Memory Section Manager 前面的加号“+”，将这个模块展开，将其中的 IDATA 和 IPROG 重新设置一下，设置如图 4-6 和图 4-7 所示。其他的存储区不用改变，这些存储区的命名是 DSP/BIOS 默认的，可以为它们改名，但是没有必要。因为本实例中没有用到外部 RAM，所以可以将 EPROG 和 EDATA 删掉，如图 4-8 所示。注意，DSP/BIOS 中 0x007c 到 0x007f 四个字是具有专门用途的存储区，用户不能占用。

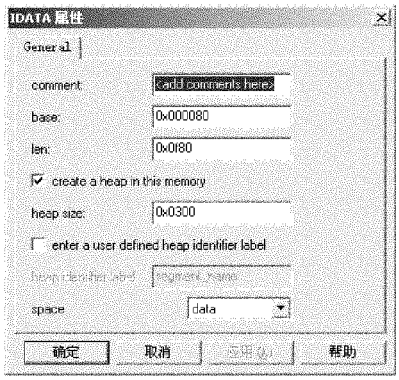


图 4-6　IDATA 设置

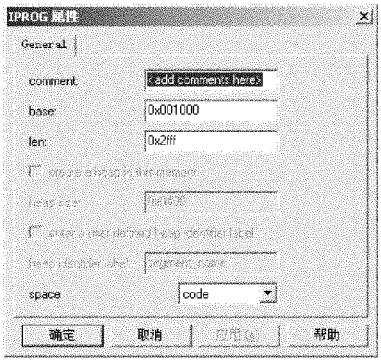


图 4-7　IPROG 设置

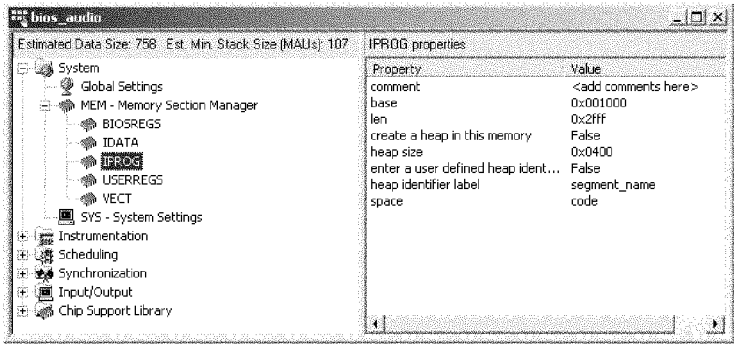


图 4-8　存储器的配置

图 4-6 和图 4-7 的文本形式如下：

PAGE 1:　IDATA:　origin=0x80,　　length=0xf80

PAGE 0:　IPROG:　origin=0x1000,　　length=0x2fff

(3) 硬件中断设置：在图 4-4 中选择 HWI-Hardware Interrupt Sevice Routine Manager 前面的加号“+”，打开 VC5402 的所有中断，如图 4-9 所示。

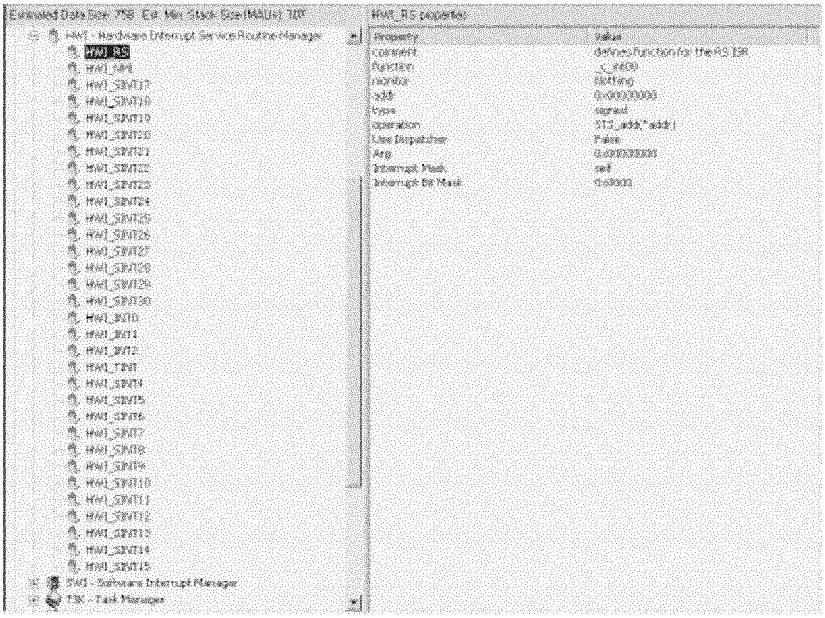


图 4-9　VC5402 的硬件中断表

在图 4-9 中选择需要设置的硬件中断，在单击右键弹出的菜单中选择属性一项，可以设置相应的中断调用函数。这个设置要在 DSP/BIOS 配置文件加入到工程文件中后才能设置。本例中没有使用中断。

(4) 多通道缓冲串行口设置：在图 4-4 中选择 McBSP-Multichannel Buffered Serial Port 前面的加号“+”，展开这个模块，如图 4-10 所示。图 4-10 中有一个串行口配置管理器和一个串行口硬件资源管理器，在配置管理器中可以加入多个串口工作配置模式，这些工作配置模式可以在资源管理器中设置为初始化的配置。本例中创建了一个 mcbspCfg1 配置，并在串口硬件资源管理器中将它设为初始化值。具体方法是：用鼠标右键单击 MCBSP Configuration Manager（配置管理器），在弹出的菜单中选择 Insert mcbspCfg 项，出现一个 mcbspCfg0，将它重新命名为 mcbspCfg1。

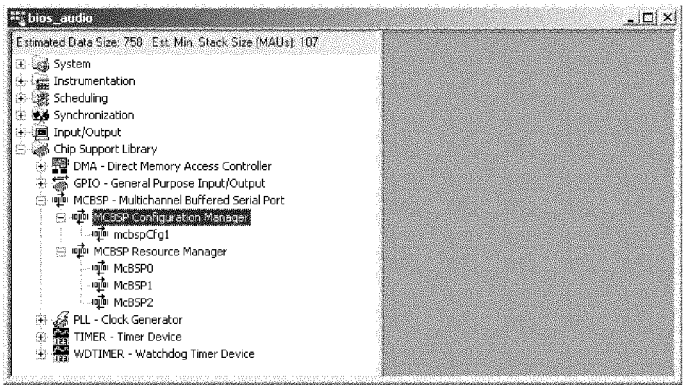


图 4-10　McBSP 串行口模块

选择 mcbspCfg1，在单击鼠标右键弹出的菜单中选择 Properties，出现如图 4-11 所示的窗口。这个窗口上有 12 个选项卡，包括了 McBSP 的所有映射寄存器的设置。其他需要重新设置的选项卡如图 4-12 至图 4-17 所示，其余的没有列出的选项卡保持不变。

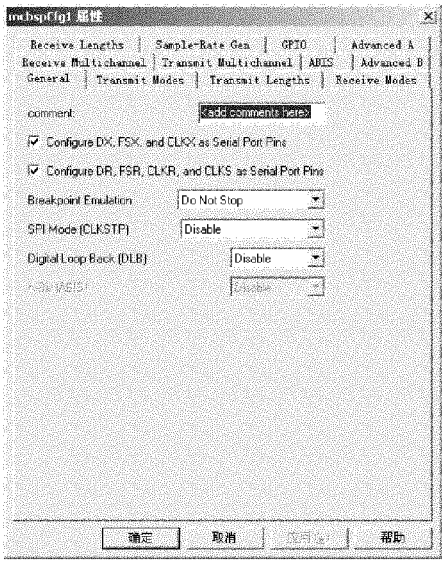


图 4-11　mcbspCfg1 属性

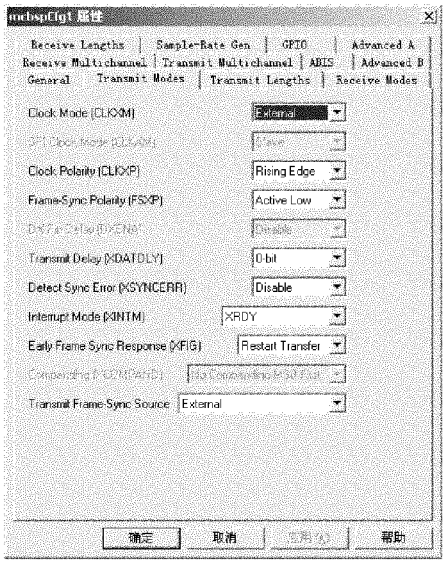


图 4-12　发送模式选项卡

图 4-11 中设置 DX、FSX、CLKX 和 DR、FSR、CLKR 作为串口控制管脚。图 4-12 设置串口的发送数据端工作在从模式，时钟来自外部 A/D 器件。当帧信号处于上升沿时，开始发送数据，帧信号低电平有效，输出数据中断由 XRDY 触发。

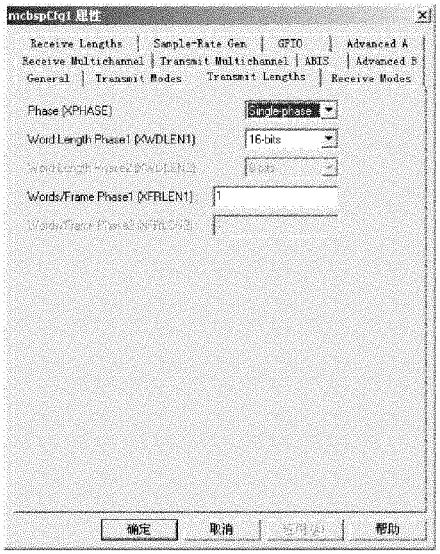


图 4-13　传送字长设置

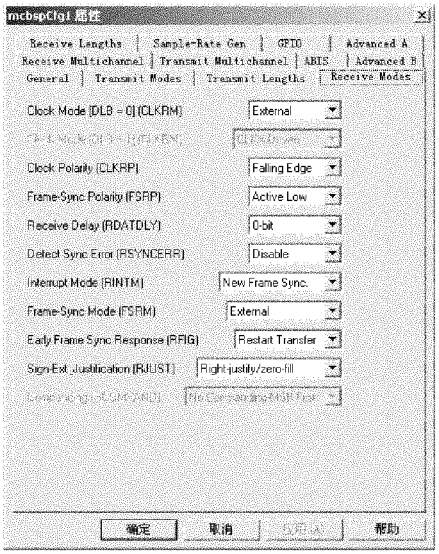


图 4-14　接收模式设置

图 4-13 设置发送的数据是字长为 16 bit 的单相数据（一帧仅有一种数据），一帧只有一个字。图 4-14 设置多通道缓冲串行口的工作模式，时钟为外部时钟源。当帧信号的下降沿开始接收数据时，帧信号低电平有效，接收数据为右对齐方式。

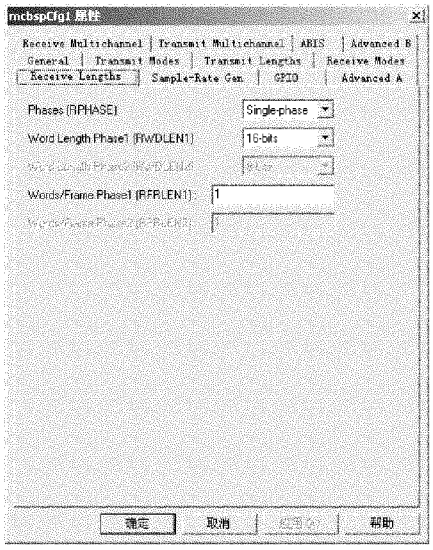


图 4-15　接收字长设置

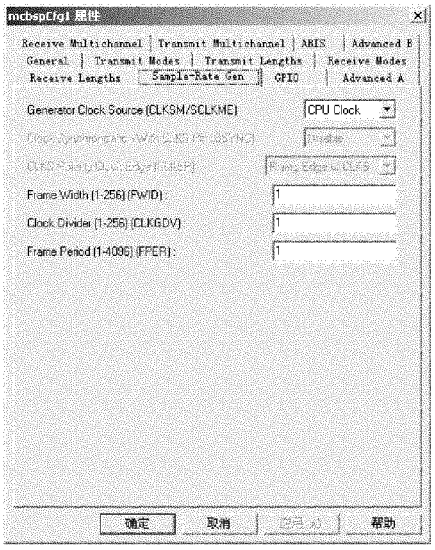


图 4-16　采样时序设置

图 4-15 将接收字长设置为每帧一个字，每字 16 bit，且为单相帧。图 4-16 设置采样时钟发生器。图 4-17 设置串行口的几个控制寄存器的值。

图 4-11 至图 4-17 设置完成后，选择 MCBSP Resource Manager 下的 McBSP1，在单击右键的弹出菜单中选择 Properties，进入图 4-18 所示的窗口。按窗口中所示将 mcbspCfg1 设置为 McBSP1 的初始化配置，点击“确定”完成设置。

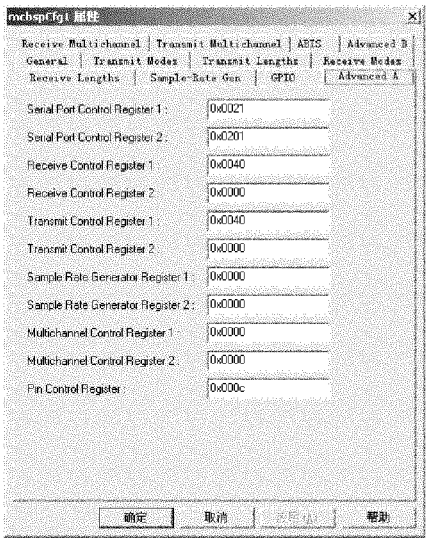


图 4-17　寄存器设置

图 4-18　McBSP1 设置

完成上面所有设置后，将文件再保存一次，关闭这个 DSP/BIOS 组件管理器。然后将刚刚保存好的文件 bios_audio.cdb 加入到工程文件中去，加入之后会自动地加入 bios_audiocfg.s54 和 bios_audiocfg_c.c 两个文件到 Generated Files 项目下。bios_audiocfg.s54 是汇编语言格式的，bios_audiocfg_c.c 是 C 语言格式的，里面包含了对于串行口 1 的初始化，这两个文件都是 DSP/BIOS 自动生成的，如没有特别需要请勿改动这些文件。除这两个文件之外，DSP/BIOS 还生成一个存储器配置文件 bios_audiocfg.cmd，这个文件中包含了在 DSP/BIOS 图形界面下的配置，很容易读懂，请也不要手工改动这个文件，另请将这个文件也加入到工程文件中去。

在 MCBSP 资源管理器中有三个 McBSP，在 VC5402 中只有两个。在本实例中只用到了 McBSP1，其他的两个是不能删除的，不用设置即可，它们不会影响程序的运行。

？第三步：加入库文件和一个空的主程序文件，完成工程文件的框架。这一步中需要向工程文件中加入 DSP/BIOS 运行支撑库 csl5402.lib（根据目标板的需要选定，这里用的是 SY-5402EVM 板）和 C 语言运行支撑库 rts.lib。特别指出，这些库文件不用手工加入到工程文件中，这并不是不需要这些库文件，而是因为在 bios_audio.cmd 文件中已经指明了库文件的路径，所以这里可以不加入运行支撑库。编写一个包含空的 main()函数的主文件 bios_audio.c，将这个文件加入到工程文件中去。完成后的项目管理器如图 4-19 所示。

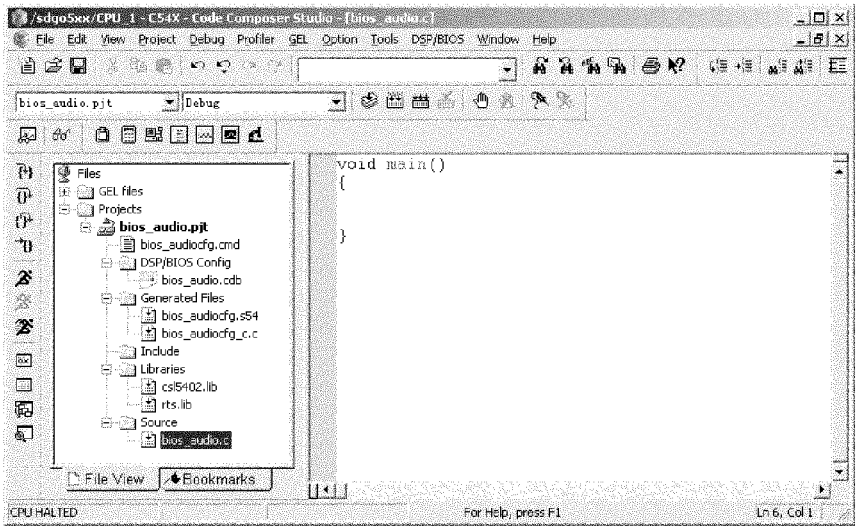


图 4-19　工程文件的框架

将这个工程文件编译一下，大量的包括文件会出现在 include 项目中，这些文件中包括了 DSP/BIOS 的全部可调用的 API 函数的原型。用户可以根据需要查看相应的文件。以 .h54 和 .s54 结尾的为汇编语言文件，以 .h 结尾的为 C 语言头文件。用户以可以通过这些 C 语言文件查看的 API 函数原型，也可以在“帮助”中查找，需要什么功能的 API 函数就只关心这些 API 函数即可，而不用全部浏览。

？第四步：编写主函数，实现所需要的功能。这一步是在上面的框架中，完善主程序文件，加入所需要的功能。在本实例中，没有用到中断，只是将 CPU 初始化，将 CSL 初始

化，并使用 CSL 的 API 函数完成对串行口的读写操作。完成后的工程文件如图 4-20 所示。

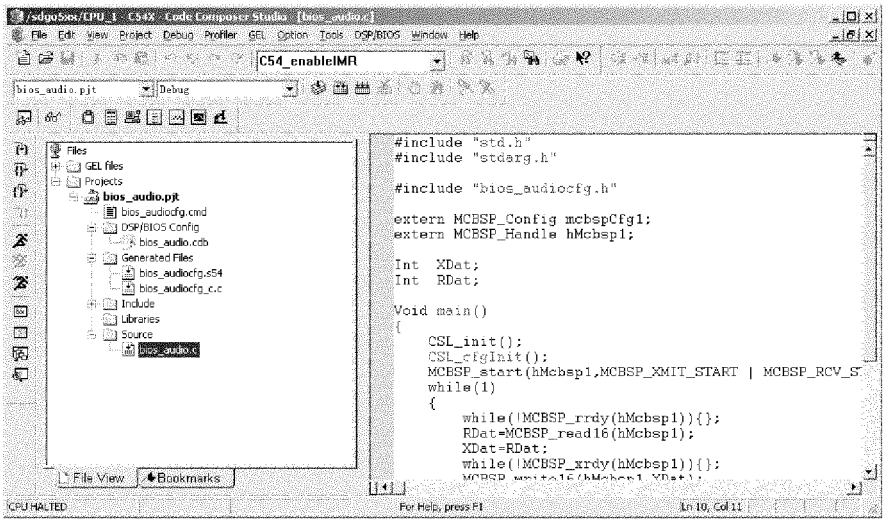


图 4-20　完整的工程文件窗口

现在将这个工程文件重新编译连接之后，就可以装入到目标板上运行了，运行效果是非常好的。但到硬盘上看一下这个目标文件的大小，会令人大吃一惊，这个几乎没有包含任何算法程序的文件竟会超过 64 KB！其实这只是表面现象。再仔细看一下，无论是 Degug 版还是 Release 版，都包括几个 .obj 文件，与主程序同名的那个只略超过 1 KB 或不足 1 KB。也就是说，真正的主程序文件变小了，大的目标文件主要是 .cdb 造成的。但是这些文件是不影响执行速度的。所以有大的存储器时，可以放心地使用 DSP/BIOS。再看一下 MAP 表文件就会得到更清楚的答案。这个 MAP 表太大，下面仅给出了头部，显示了程序占用存储器的情况。但是采用 Boot Loader 装入程序时，也是完全按这个 MAP 表装入的。一般会用到外部 RAM 的映射存储器存放 DSP/BIOS 的其他信息。

```
******************************************************************************
TMS320C54x COFF Linker                PC Version 3.70
******************************************************************************
>> Linked Sun Dec 22 17:47:19 2002

OUTPUT FILE NAME:    <./Debug/bios_audio.out>
ENTRY POINT SYMBOL: "_c_int00"    address: 0000017f

MEMORY CONFIGURATION

         name            origin          length          used            attr        fill
         ----------------  --------        ---------       --------        ----        --------
PAGE  0: VECT            00000080        00000080        00000080        RWIX
```

	IPROG	00000100	00002000	00001aaa	RWIX
	EPROG	00008000	00003f80	00000000	RWIX
PAGE 1:	USERREGS	00000060	0000001c	00000000	RWIX
	BIOSREGS	0000007c	00000004	00000004	RWIX
	IDATA	00002100	00001f00	00000adc	RWIX
	EDATA	00008000	00008000	00000000	RWIX

在这个工程文件中，用户所写的代码都在主文件 bios_audio.c 中，其他的操作就是配置 DSP/BIOS。可见，程序设计的重点完全集中在算法研究上去了，所以说 DSP/BIOS 是一种优秀的程序设计方法。下一节给出了这个工程文件中全部的手工代码，并进一步剖析一个真正的 DSP/BIOS 程序的全貌。

4.2.3　源程序清单和 DSP/BIOS 编程分析

1) bios_audio.c 文件

下面列出了上一节的工程文件中的主程序文件 bios_audio.c 的全部内容。

```
#include "std.h"
#include "stdarg.h"

#include "bios_audiocfg.h"

extern MCBSP_Config mcbspCfg1;
extern MCBSP_Handle hMcbsp1;

Int   XDat;
Int   RDat;

Void main()
{
    //BIOS_init();
    CSL_init();                    //可以省略，但需要加入一个 CPU 和串口的初始化程序
    CSL_cfgInit();                 //可以省略，但需要加入一个串口初始化程序
    MCBSP_start(hMcbsp1,MCBSP_XMIT_START | MCBSP_RCV_START,0x300u);
    while(1)
    {
        while(!MCBSP_rrdy(hMcbsp1)){};   //调用 API 函数实现读写串口的操作
        RDat=MCBSP_read16(hMcbsp1);
        XDat=RDat;
        while(!MCBSP_xrdy(hMcbsp1)){};
```

```
        MCBSP_write16(hMcbsp1,XDat);

    }
    //MCBSP_close(hMcbsp1);
    //BIOS_start();  隐式调用
    return;
}
```

从这个主程序中可以看出，本程序中实际需要编写的代码很少。本实例仅是使用了 CSL 库进行设计的，在没有使用中断的情况下，可以手工配置 CSL 库。一个完全意义上的 DSP/BIOS 程序是使用中断来完成的。

下面列出了一个 DSP/BIOS 程序的工作情况，这个工作情况按执行顺序分为五步：

2) DSP/BIOS 程序工作过程

? 第一步：硬件上电复位 DSP。在这个过程中，程序指针 PC 指向 c_int00，堆栈指针 SP 指向栈底，状态寄存器完成初始化，.cint 段完成初始化。

? 第二步：调用 BIOS_init()函数初始化 DSP/BIOS 的各个模块。各个模块的初始化函数 MOD_init（MOD 指代模块名，例如，对硬件中断为 HWI）会被自动调用，并完成初始设置，BIOS_init()函数是由 DSP/BIOS 自动生成的。

? 第三步：调用主函数。因为这时硬中断和软中断均被屏蔽了，所以这个 main()函数里可以进行硬件的一些初始化工作，还可以放置除死循环之外的任何程序段。常常使用 DSP/BIOS 的用户可能会发现，这个主函数只是做一些初始化的工作，真正的数字信号处理算法是由中断函数调用的，而且在 DSP/BIOS 编程中，不能使用关键字 interrupt 或 INTERRUPT。这个主函数是用户设计的，其他的都是 DSP/BIOS 图形界面设计自动生成的。

? 第四步：调用 BIOS_start()进入 DSP/BIOS 工作状态。BIOS_start()函数也是由 DSP/BIOS 自动生成的，它调用各个模块的 MOD_startup()函数，并激活各个模块，包括中断。

? 第五步：调用 IDL_loop()使 DSP 进入空闲状态，等待中断信号的到来。IDL 状态可以实现主机对目标板上 DSP 的访问，如果遇到中断，将首先执行中断任务。进入 IDL 状态是真正的信号处理工作的开始，同时各种 DSP 的示波器也可以访问这个状态。

从上面五步可以看出，DSP/BIOS 程序设计中不再有死循环。更重要的是，主程序的功能被消弱，功能的实现是通过一些中断函数来完成的。可能读者在看一些 DSP/BIOS 程序时不能理解一些文件里的函数没有显式的调用格式，而主程序又没有什么功能，这个程序是如何工作的？其实，它强大的功能就是那些函数来完成的，这些函数在 DSP 处于 IDL 状态时被中断触发执行。

为了进一步说明这个问题，下一节中使用中断的方法重写了上面的程序实例，这是一个真正意义上的 DSP/BIOS 程序设计实例。这个实例是仿写了 CCStudio 提供的一个 DSP/BIOS 实例，同时加入了注解，以此说明了一些“示波器”的用法。该程序在 SY-5402EVM 板上调试通过，运行效果与前面的程序完全相同。

4.2.4　DSP/BIOS 中断编程

这段程序是仿写 CCStudio 提供的一个源程序，为了便于理解，对程序进行了适当的简化，有几个子程序写得相当优秀而又有代表性，此处将原文引用并作了注明，其他的程序是另外写的。为了便于阅读，选用了与原例程相同的变量符号和函数名来编写。

DSP/BIOS 是模块化的程序设计方法，有些程序的引用是没有显式定义的，必须到 DSP/BIOS 配置文件中去找它们的入口点。下面分了三部分介绍这个程序。第一部分给出程序的工作原理；第二部分给出程序中所用的 DSP/BIOS 配置文件；第三部分将原程序代码全部列出，并作了详细注解。

1．工作原理

这个程序的工作过程是：由 DSP 的多通道串行口 1（McBSP1）的接收数据硬中断（对于 VC5402 是 HWI_SINT10 中断）激活中断服务程序，这个中断服务程序访问 DSP/BIOS 的数据管道组件（PIP）。访问这个 PIP 对象的方法是通过调用 PIP 的 API 函数，再通过 API 函数的调用进一步激活软中断 audioSWI，最后借助软中断完成数据流的读入和写出。硬中断的优先级总是高于软中断的优先级。

以上是基本的工作过程，也是这个程序工作的一根红线，必须把握住。下面详细论述。

DSP/BIOS 提供了一个称为 PIP 的组件，这个组件类似一条流过两个水库的管道一样，这两个水库可以被明确地指定容量，所以把 PIP 组件称为数据缓冲通道组件。PIP 通道具有一个写入数据端和读出数据端，而且只有当数据完全为空时才能写入，当数据充满时才能读出。通过调用 PIP 的 API 函数 PIP_get 检测 PIP 管道中是否数据已满，如果已满，则自动调用 notifyReader 指向的软中断函数；通过调用 API 函数 PIP_alloc 判断 PIP 管道是否已空，如果已经为空，则自动调用 notifyWriter 指向的软中断函数。可见，PIP 可以节约 CPU 的工作时间。

这个 PIP 对象在本程序中是受到两个中断线程控制的，一个是来自 McBSP 串行口的硬中断 HWI，另一个是供助 DSP/BIOS 定义的软中断 SWI，它们分别控制了 PIP 管道的读与写，而且也只能控制一端。因为程序最终完成的任务是：从 McBSP1 数据输入端接收来数据，又将数据回送至 McBSP1 数据发送端，所以至少需要建立两个 PIP 管道。

软中断是 DSP/BIOS 特有的，它的工作原理同硬中断相似，当软中断被激活时，它会强占比它优先级低的中断及空闲状态的 CPU 时隙，当它运行完毕后再归还 CPU 时隙。软中断的优先级可以在 DSP/BIOS 中设定，但是它的优先级始终比硬中断低。中断发生后，程序指针被强制转移到中断地址入口点，并且中断前的环境常常被保存在椎栈中。这个指向的程序被称为中断服务程序，这个程序的功能是需要用户编程指定的。

本程序工作的原理框图和程序调用流程图如图 4-21 所示。

从图 4-21 中看不到主程序的影子。事实上，主程序的作用是微不足道的，这是 DSP/BIOS 程序设计的一大特色，有些面向对象或实体进行程序设计的味道。

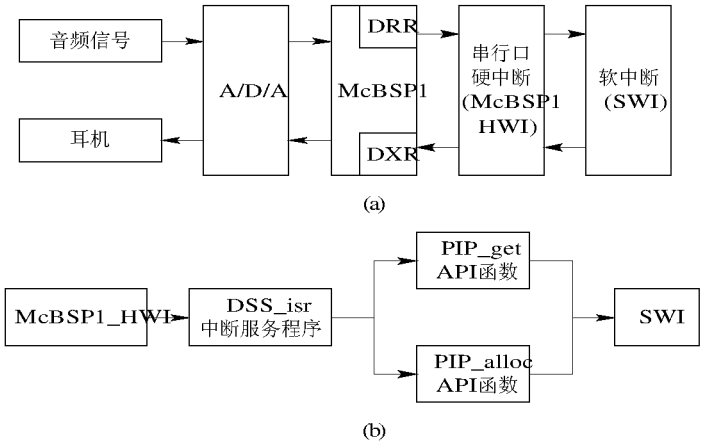


图 4-21　中断程序的工作原理框图和调用流程图

(a) 工作原理框图；(b) 调用流程图

2．DSP/BIOS 配置

下面用图解的方法给出了本程序中 DSP/BIOS 的配置。与前面程序例程中的 DSP/BIOS 配置方法相似，这里直接从 DSP/BIOS 组件管理器（直译为 DSP/BIOS 配置工具）开始，按下面的介绍一步一步进行配置。

? 第一步：新建一个工程文件，工程文件名为 new_audio.pjt。新建一个 DSP/BIOS 配置文件，命名为 new_audio.cdb。

? 第二步：设置 System/Global Settings 如图 4-22 所示。注意，SY-5402EVM 板的 CLKMD 的设置应该改动一下，具体改动同前。

? 第三步：使用 MCBSP Configuration Manager 新建一个 DSS_mcbspCfg0，方法同前。将这个配置赋给 MCBSP1，如图 4-23 所示。

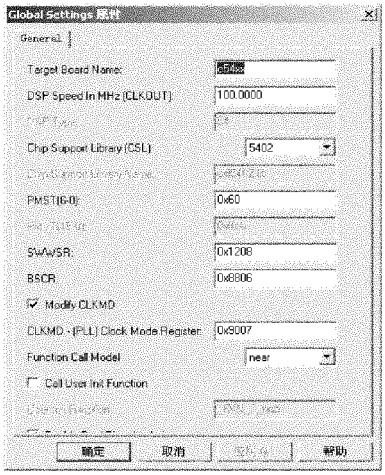


图 4-22　CPU 配置

图 4-23　MCBSP1 属性

? 第四步：新建两个数据管道 DSS_rxPipe 和 DSS_txPipe，如图 4-24 所示。

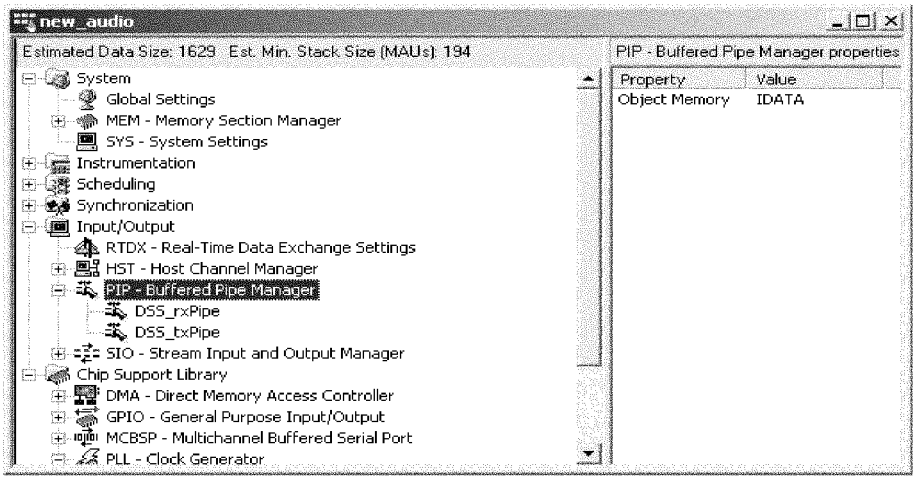


图 4-24 两个数据 PIP 通道

这个通道的属性设置如图 4-25 和图 4-26 所示，通过鼠标右键弹出菜单设置，方法同前。

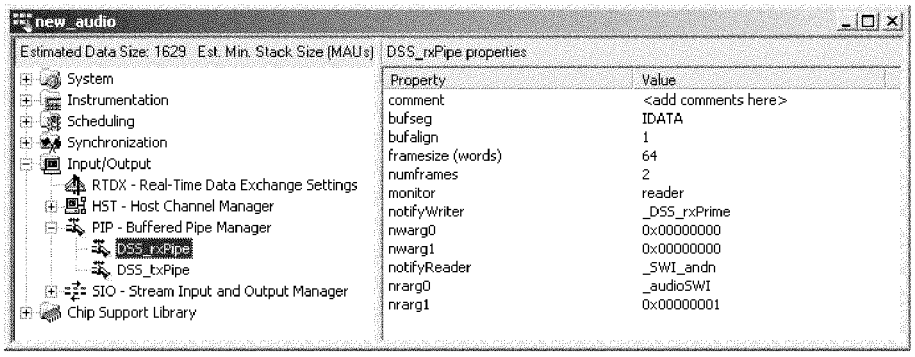


图 4-25　DSS_rxPipe 属性

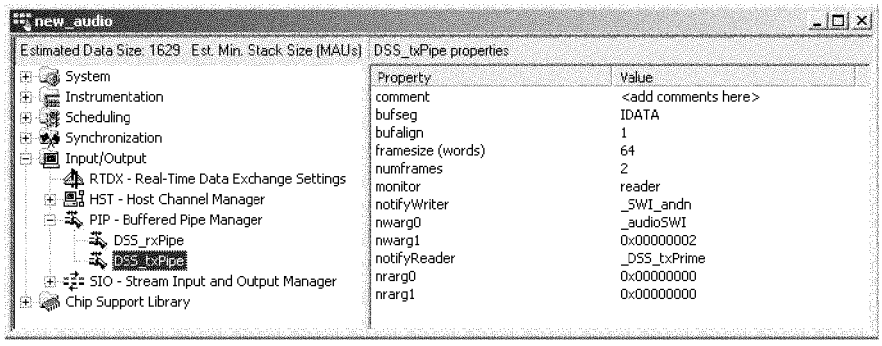


图 4-26　DSS_txPipe 属性

? 第五步：设置串口的中断，如图 4-27 所示。

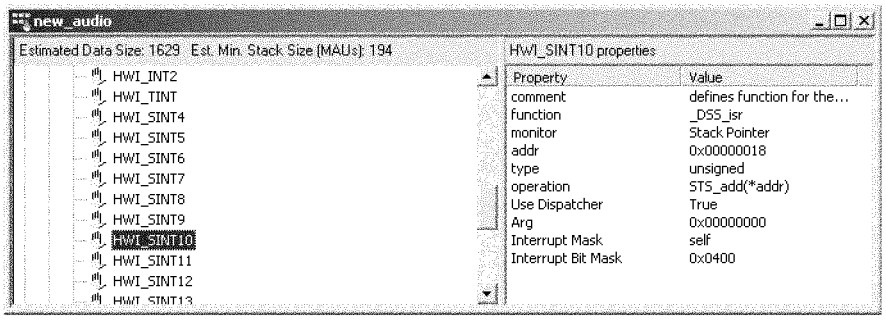


图 4-27 硬中断设置

? 第六步：添加并设置一个软中断 audioSWI，如图 4-28 所示。

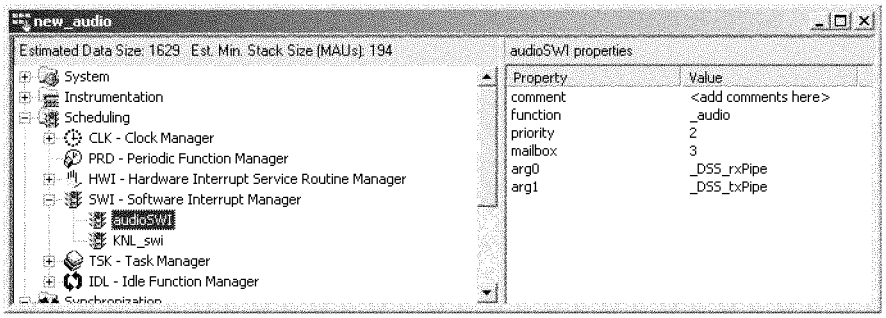


图 4-28 软中断设置

在图 4-28 中必须设置 mailbox，触发软中断时会判断 mailbox 的值，当 mailbox 自减到 0 时会执行软中断。arg0 和 arg1 是软中断的两个指针类型的参数。

? 第七步：完成以上几步后，将配置文件再保存一次，并将它加入到 new_audio.pjt 工程文件中。

3．工程文件源程序

下面列出工程文件中所需要的其他文件，它们是：new_audio.h，audio.c，dss_cisr.c 和 dss_face.c，这些文件全部用 C 语言写成。在 DSP/BIOS 编程中可以完全使用 C 语言进行程序设计（而在第三章程序中的中断向量表文件则是用汇编语言编写的）。注意，中断服务函数不能加上关键字 interrupt 或 INTERRUPT，DSP/BIOS 有专用的访问中断（进入和退出中断）API 函数。

```
//下面是文件 new_audio.h
#include <std.h>        //必须包括在最前面，其中定义了 DSP/BIOS 的数据类型
#include <pip.h>
#include <log.h>
```

```
#include <hst.h>
#include <clk.h>
#include <sts.h>
#include <trc.h>
#include <string.h>
#include <stdarg.h>

#define CHIP_5402    1
#include <csl_mcbsp.h>

#define DSS_RXERR    0x1
#define DSS_TXERR    0x2

extern far LOG_Obj trace;

extern Void audio(PIP_Obj *in, PIP_Obj *out);          //下面为声明的外部变量或函数

extern far Int DSS_error;
extern Void DSS_init(Void);
extern far PIP_Obj DSS_rxPipe;                         //数据管道
extern far PIP_Obj DSS_txPipe;

extern far MCBSP_Handle DSS_hMcbsp0;
extern far STS_Obj DSS_ioPhase;

extern Void DSS_rxPrime(Bool calledByISR);             //数据管道的读写操作
extern Void DSS_txPrime(Bool calledByISR);
```

下面为 dss_face.c 文件

```
// By ZhnYong@21cn.com
// All rights reserved

#define SPSA_ADDR(port)          (port ? 0x48 : 0x38)
#define SPSD_ADDR(port)          (port ? 0x49 : 0x39)

#define DRR2_ADDR(port)          (port ? 0x40 : 0x20)
#define DRR1_ADDR(port)          (port ? 0x41 : 0x21)
#define DXR2_ADDR(port)          (port ? 0x42 : 0x22)
```

```
#define DXR1_ADDR(port)          (port ? 0x43 : 0x23)

#define   SPCR1_SUBADDR     0x00
#define   SPCR2_SUBADDR     0x01
#define   RCR1_SUBADDR      0x02
#define   RCR2_SUBADDR      0x03
#define   XCR1_SUBADDR      0x04
#define   XCR2_SUBADDR      0x05
#define   SRGR1_SUBADDR     0x06
#define   SRGR2_SUBADDR     0x07
#define   MCR1_SUBADDR      0x08
#define   MCR2_SUBADDR      0x09
#define   RCERA_SUBADDR     0x0A
#define   RCERB_SUBADDR     0x0B
#define   XCERA_SUBADDR     0x0C
#define   XCERB_SUBADDR     0x0D
#define   PCR_SUBADDR       0x0E

#define          bsp_SPCR11          0x0021
#define          bsp_SPCR21          0x0201
#define          bsp_PCR1            0x000C
#define          bsp_RCR11           0x0040
#define          bsp_RCR21           0x0000
#define          bsp_XCR11           0x0040
#define          bsp_XCR21           0x0000
#define          bsp_SRGR11          0x0000
#define          bsp_SRGR21          0x0000
#define          bsp_MCR11           0x0000
#define          bsp_MCR21           0x0000

void DSS_init(void)                  //下面给出了一种串口初始化的方法
{
    *(int *)(0x0007)&=0xBFFF;//   CPL=0
    *(int *)(0x0006)&=0xFE00;//   DP=0

    *(int *)(0x0007)|=0x0900;// INTM=0;SXM=0

    *(int *)(0x0054)&=0xff3f;// DMPREC[6,7] clear 00 interrupts for McBSP
```

```
    *(volatile int *)DXR1_ADDR(1)=0x00;

    *(volatile int *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //select SPCR11
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR11 & 0xFFFE;
    *(volatile int *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR21 & 0xFFFE;
    *(volatile int *)SPSA_ADDR(1)=RCR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_RCR11;
    *(volatile int *)SPSA_ADDR(1)=RCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_RCR21;
    *(volatile int *)SPSA_ADDR(1)=XCR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_XCR11;
    *(volatile int *)SPSA_ADDR(1)=XCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_XCR21;
    *(volatile int *)SPSA_ADDR(1)=SRGR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SRGR11;
    *(volatile int *)SPSA_ADDR(1)=SRGR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SRGR21;
    *(volatile int *)SPSA_ADDR(1)=MCR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_MCR11;
    *(volatile int *)SPSA_ADDR(1)=MCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_MCR21;
    *(volatile int *)SPSA_ADDR(1)=PCR_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_PCR1;

    *(volatile int *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //Enable McBSP1
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR11;
    *(volatile int *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR21;

    *(volatile int *)(0x0000) |= 0x0400;      //IMR open Mcbsp1 receive
    *(volatile int *)(0x0001)=0xffff;         //IFR

    *(int *)(0x0007)|=0x4000;                 // CPL=1

    DSS_rxCnt = (sizeof(Int) / sizeof(MdInt)) * PIP_getWriterSize(rxPipe);
    DSS_txCnt= (sizeof(Int) / sizeof(MdInt)) * PIP_getReaderSize(txPipe);
}
```

//下面为 audio.c 文件，其中包括了主程序。读者可以从中看到这个主程序的功能是有限的，在 DSP/BIOS 程序设计中，也是可以被替代的。

```
//ZhangYong.
//The    mimic of TI audio.pjt
#include "new_audio.h"
//main Program finishs the initialization
Void main()
{
      DSS_init();   /* Initialize serial port 1 */
      LOG_printf(&trace, "Initialization finished!\nAudio started during BIOS idle!!\n");
      return;          /* BIOS idle ;Waiting HWI and SWI     进入了 IDLE 状态*/
}

//Function audio() Called by    audioSWI
//软中断调用该函数，完成数据在两个数据管道之间的交换
Void audio(PIP_Obj *in, PIP_Obj *out)
{
      Uns *src, *dst;
      Uns size;

      if (PIP_getReaderNumFrames(in) == 0 || PIP_getWriterNumFrames(out) == 0)
      {
            LOG_printf(&trace,"Error: Missed the PIP pointer");
            return;
      }
      /* get input data and allocate output buffer */
      PIP_get(in);
      PIP_alloc(out);

      src=PIP_getReaderAddr(in);
      dst=PIP_getWriterAddr(out);
      size=PIP_getReaderSize(in);
      PIP_setWriterSize(out,size);

      for(; size>0;size--)
      {
            *dst++=*src++;
      }
```

```
    /* check for real-time error */
    if (DSS_error != 0)
    {
        LOG_printf(&trace,
        "Error: Program missed real-time! (DSS_error = 0x%x)", DSS_error);
        DSS_error = 0;
    }

    /* output data and free input buffer */
    PIP_put(out);
    PIP_free(in);
}
```

//下面为 DSS_cisr.c 文件，包括了硬中断调用函数。这个程序的写法是相当经典的一种程序设计方法，这里原文引入，略做了一点修改（作者觉得原文中的一处 if 语句用得不恰当，将其改为了 while 语句）。

```
//Filename:dss_cisr.c
//ZhangYong,All rights reserved.

#include "new_audio.h"

Int     DSS_error = 0;

Int     DSS_rxCnt = 0;    //接收管道帧长
Int     DSS_txCnt = 0;    //发送管道帧长

Int     *DSS_rxPtr = NULL;
Int     *DSS_txPtr = NULL;
typedef struct {
    Bool  enable;
} DSS_Obj;
DSS_Obj    DSS_config = {0};                    /* enable tracing */

// DSS_isr() interrupt service program Called by HWI_SINT10
//下面为硬中断服务程序。完成数据管道与串口 1 之间的数据交换
void DSS_isr(void)
{
```

```
volatile int dummy;
int rxDone = 0;                 //接收管道满标志
int txDone = 0;                 //发送管道空标志，“空”是指没有有效的发送数据了

if (DSS_rxCnt)                  //接收管道从串口 1 接收数据直到满为止
{
    *DSS_rxPtr++ = DRR1(DSS_hMcbsp0);
    DSS_rxCnt--;
    if (DSS_rxCnt == 0)
    {
        rxDone = 1;
    }
}
else
{
    dummy = DRR1(DSS_hMcbsp0);
    DSS_error |= 0x1;
}

if (DSS_txCnt)          //发送管道发送数据到串口 1 直到发送管道空为止
{
    DXR1(DSS_hMcbsp0) = *DSS_txPtr++ & 0xfffe;
    DSS_txCnt--;
    if (DSS_txCnt == 0)
    {
        txDone = 1;
    }
}
else
{
    DXR1(DSS_hMcbsp0) = 0;
    DSS_error |= 0x2;
}

if ((rxDone | txDone) == 0)
{
    return;
}
```

```
    if (rxDone)                    //接收通道满后调用 DSS_rxPrime()
    {
        /*
        * don't have to set writerSize or writerAddr
        * since we only provide "full" frames and these
        * fields are already set.
        */
        PIP_put(&DSS_rxPipe);
        DSS_rxPrime(TRUE);
    }

    if (txDone)                    //发送通道空时调用 DSS_txPrime()
    {
        /*
        * don't have to set readerSize or readerAddr
        * since they're already set.
        */
        PIP_free(&DSS_txPipe);
        DSS_txPrime(TRUE);
    }
}

/*
*   ======= DSS_txPrime ========
*   Called when DSS_txPipe has a full buffer to be transmitted
*   (i.e., when notifyReader() is called) and when the DSS ISR
*   is ready for more data.
*/
void DSS_txPrime(Bool calledByISR)          //调用软中断更新发送通道
{
    PIP_Obj     *txPipe = &DSS_txPipe;
    int delay = 10;          /* or 2, 3, etc. */
    static Int nested = 0;

    LOG_message("DSS_txPrime(): %u", CLK_gethtime());

    if (nested)
    {           /* prohibit recursive call via PIP_get() */
        return;
```

```
}

while (delay)
{       /* ensure that output does not start too soon */
    delay--;
    //return;
}

nested = 1;
//
if (DSS_txCnt == 0 && PIP_getReaderNumFrames(txPipe) > 0)
{
    Int count, i;
    MdInt *dst;// typedef short MdInt
    PIP_get(txPipe);

    /* must set 'Ptr' before 'Cnt' to synchronize with isr() */
    DSS_txPtr = PIP_getReaderAddr(txPipe);

    /* ensure bit 0 of DAC word is 0; AIC uses bit 0 to request control */
    count = (sizeof(Int) / sizeof(MdInt)) * PIP_getReaderSize(txPipe);
    for (i = count, dst = (MdInt *)DSS_txPtr; i > 0; i--)
    {
        *dst++ = 0xfffe & *dst;    //AD50 required
    }

    DSS_txCnt = count;

}
else if (calledByISR && (DSS_error & DSS_TXERR) == 0)
{
        if (DSS_config.enable)
    {
        TRC_disable(TRC_GBLTARG);
    }
    LOG_error("transmit buffer underflow: %u", CLK_gethtime());
    DSS_error |= DSS_TXERR;
}
nested = 0;
```

```
}

/*
*   ======= DSS_rxPrime ========
*   Called when DSS_rxPipe has an empty buffer to be filled;
*   e.g., when notifyWriter() is called) and when the DSS ISR
*   is ready to fill another buffer.
*/
void DSS_rxPrime(Bool calledByISR)                    //调用软中断更新接收通道
{
    PIP_Obj      *rxPipe = &DSS_rxPipe;
    static Int   nested = 0;

    LOG_message("DSS_rxPrime(): %u", CLK_gethtime());

    if (nested)
    {           /* prohibit recursive call via PIP_alloc() */
        return;
    }
    nested = 1;

    if (DSS_rxCnt == 0 && PIP_getWriterNumFrames(rxPipe) > 0)
    {
        PIP_alloc(rxPipe);

        /* must set 'Ptr' before 'Cnt' to synchronize with isr() */
        DSS_rxPtr = PIP_getWriterAddr(rxPipe);
        DSS_rxCnt = (sizeof(Int) / sizeof(MdInt)) * PIP_getWriterSize(rxPipe);

    }
    else if (calledByISR && (DSS_error & DSS_RXERR) == 0)
    {
        if (DSS_config.enable)
        {
            TRC_disable(TRC_GBLTARG);
        }
        LOG_error("receive buffer overflow: %u", CLK_gethtime());
        DSS_error |= DSS_RXERR;
}
```

```
        nested = 0;
    }
    //上面程序中借用了作者仿写的程序的英文注解
```

4．DSP/BIOS 软“示波器”

DSP/BIOS 自带了七个软“示波器”，其中一个是控制面板。图 4-29 为本程序运行的实时分析图。因为加入了这些实时分析窗口，计算机用户可能会感到计算机反应减慢，这与仿真器也有关系（作者用的是 XDS510，可能用 XDS560 效果会好一些），但是 DSP 程序运行是不会受到影响的。

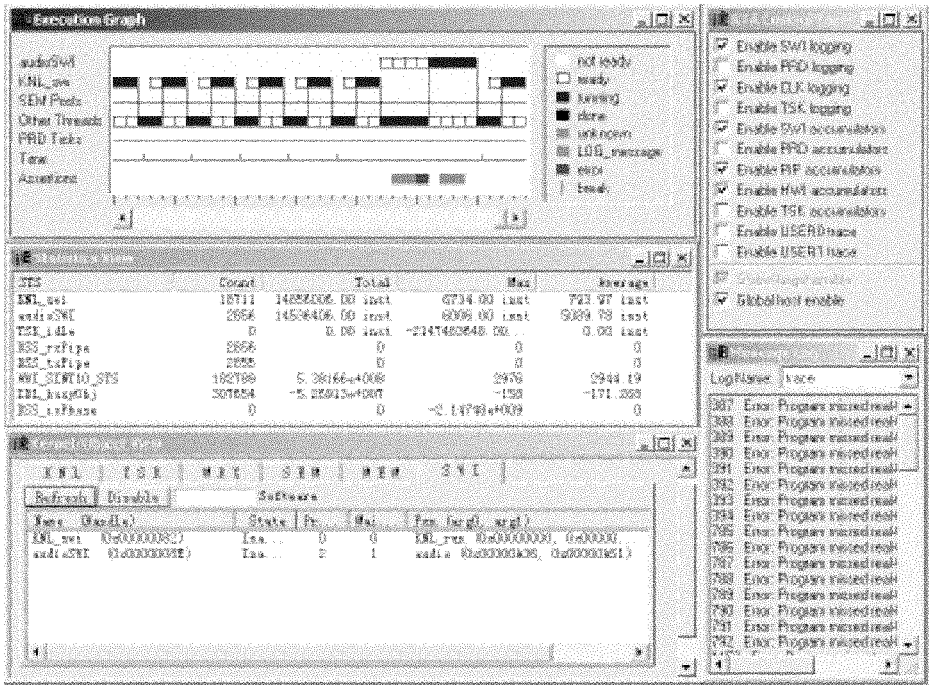


图 4-29　DSP/BIOS 的实时分析结果真的一部分

在图 4-28 中没有主机通道控制器窗口。下面依次说明图 4-28 中各个窗口的含义。

右上角的 RTA Control Panel 栏是用来定义实时分析的项目的，通过复选框来定义需要查看的实时项目，缺省值为全部选中。右下角为实时信息窗口，这个信息窗口中的信息与左上角的实时运行图样窗口中的 LOG Message 图例是严格对应的，即当在实时运行图样窗口出现一个 LOG Message 图例时，则实时信息窗口会出现这条信息。左上角的实时运行图形窗口样窗口使用不同的图例表示不同的线程占有 CPU 的情况，对于黑白打印机看不出来图样，读者可以自己试着分析一下，从这个图至少可以看出 CPU 有一半的时间是空闲的。左边中间一个图形窗口为实时统计结果图，从中可以看出各个例程被 DSP 调用的情况。左下角的实时分析工具展示了 DSP/BIOS 内部各个组件的运行状态及情况。这些“示波器”综合使用才能有效地把握 DSP 的实时工作情况。

5．程序简化

为了深入观察 DSP/BIOS 编程的思想，可将上面的实例作一简化，就是不设置软中断和 PIP 管道，直接用串行口的硬中断调用中断服务程序，在这个中断服务程序中完成对串口的读写操作。

具体操作如下：

？第一步：删除软中断和 PIP 管道，其他部分保留，将 HWI_SINT10 的中断入口地址改为_audio，因为中断入口地址是以汇编语言形式存在的，所以必须在 C 语言的函数名前加上一个下划线。然后，保存这个 DSP/BIOS 配置文件。

？第二步：将程序调整为以下三个程序：new_audio.h，audio.c 和 dss_face.c，程序清单列在下面，可以看到主程序依然是辅助性的，真正的算法将由中断服务程序来完成。

// 下面是修改后的 new_audio.h

```
#include <std.h>
#include <pip.h>
#include <log.h>
#include <hst.h>
#include <clk.h>
#include <sts.h>
#include <trc.h>
#include <string.h>
#include <stdarg.h>

#define CHIP_5402    1
#include <csl_mcbsp.h>
//串口的地址宏定义
#define SPSA_ADDR(port)          (port ? 0x48 : 0x38)
#define SPSD_ADDR(port)          (port ? 0x49 : 0x39)

#define DRR2_ADDR(port)          (port ? 0x40 : 0x20)
#define DRR1_ADDR(port)          (port ? 0x41 : 0x21)
#define DXR2_ADDR(port)          (port ? 0x42 : 0x22)
#define DXR1_ADDR(port)          (port ? 0x43 : 0x23)

#define    SPCR1_SUBADDR          0x00
#define    SPCR2_SUBADDR          0x01
#define    RCR1_SUBADDR           0x02
#define    RCR2_SUBADDR           0x03
#define    XCR1_SUBADDR           0x04
#define    XCR2_SUBADDR           0x05
```

```
#define    SRGR1_SUBADDR        0x06
#define    SRGR2_SUBADDR        0x07
#define    MCR1_SUBADDR         0x08
#define    MCR2_SUBADDR         0x09
#define    RCERA_SUBADDR        0x0A
#define    RCERB_SUBADDR        0x0B
#define    XCERA_SUBADDR        0x0C
#define    XCERB_SUBADDR        0x0D
#define    PCR_SUBADDR          0x0E
```

//下面为串口的初始化值

```
#define        bsp_SPCR11        0x0021
#define        bsp_SPCR21        0x0201
#define        bsp_PCR1          0x000C
#define        bsp_RCR11         0x0040
#define        bsp_RCR21         0x0000
#define        bsp_XCR11         0x0040
#define        bsp_XCR21         0x0000
#define        bsp_SRGR11        0x0000
#define        bsp_SRGR21        0x0000
#define        bsp_MCR11         0x0000
#define        bsp_MCR21         0x0000

extern far LOG_Obj trace;
extern Void audio();
extern Void DSS_init(Void);
extern far MCBSP_Handle DSS_hMcbsp0;
```

//下面是修改后的 dss_face.c

```
// By ZhangYong@21cn.com
// All rights reserved

#include "new_audio.h"
void DSS_init(void)                    //对 CPU 和串口进行初始化
{
    *(int *)(0x0007)&=0xBFFF;          //CPL=0
    *(int *)(0x0006)&=0xFE00;          //DP=0

    *(int *)(0x0007)|=0x0900;          // INTM=0;SXM=0
```

```
    *(int *)(0x0054)&=0xff3f;// DMPREC[6,7] clear 00 interrupts for McBSP

    *(volatile int *)DXR1_ADDR(1)=0x00;

    *(volatile int *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //select SPCR11
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR11 & 0xFFFE;
    *(volatile int *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR21 & 0xFFFE;
    *(volatile int *)SPSA_ADDR(1)=RCR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_RCR11;
    *(volatile int *)SPSA_ADDR(1)=RCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_RCR21;
    *(volatile int *)SPSA_ADDR(1)=XCR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_XCR11;
    *(volatile int *)SPSA_ADDR(1)=XCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_XCR21;
    *(volatile int *)SPSA_ADDR(1)=SRGR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SRGR11;
    *(volatile int *)SPSA_ADDR(1)=SRGR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SRGR21;
    *(volatile int *)SPSA_ADDR(1)=MCR1_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_MCR11;
    *(volatile int *)SPSA_ADDR(1)=MCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_MCR21;
    *(volatile int *)SPSA_ADDR(1)=PCR_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_PCR1;

    *(volatile int *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //Enable McBSP1
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR11;
    *(volatile int *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile int *)SPSD_ADDR(1)=bsp_SPCR21;

    *(volatile int *)(0x0000) |= 0x0400;   //IMR open Mcbsp1 receive
    *(volatile int *)(0x0001)=0xffff;      //IFR

    *(int *)(0x0007)|=0x4000;// CPL=1
}
```

//下面是修改后的 audio.c，包括主函数和中断服务程序 audio

```
//ZhangYong.

#include "new_audio.h"

//main Program finishs the initialization
Void main()
{
    DSS_init();   /* Initialize serial port 1 */
    LOG_printf(&trace, "Initialization finished!\nAudio started during BIOS idle!!\n");
    return;       /* BIOS idle ;Waiting HWI and SWI */
}

//Function audio() Called by HWI
Void audio()          //响应串口接收数据硬中断进行串口的读写
{
    Int in_data,out_data;
    *(volatile int *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //Receive Data from McBSP1
    while(!((*(volatile int *)SPSD_ADDR(1)) & 0x0002)){};
    in_data=*(volatile int*)DRR1_ADDR(1);
    in_data &= 0xFFFE;                  // A/D/A 要求 DSP 接收数据时最后一位无效

    out_data=in_data & 0xFFFE;      // A/D/A 要求 DSP 发送数据时最后一位必须为 0
    *(volatile int *)SPSA_ADDR(1)=SPCR2_SUBADDR;   // Transmit Data To McBSP1
    while(!((*(volatile int *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile int *)DXR1_ADDR(1)=out_data;
}
```

? 第三步：编译连接生成可执行的.out 文件，装入到目标板上即可。

本程序也是在 SY-5402EVM 上调试通过的，运行结果与前面的程序相同。

图 4-30 是在程序运行时摄取下来的图形。这两个实时分析“示波器”分析显示了该程序的负载及各线程的工作情况。

细心的读者可能会发现，图 4-30 比图 4-29 稳定得多，而且 CPU 负载小得多，实际编程时，建议采用后者。但并不是前一个程序不好，只是考虑到理解方便，在仿写时作了大量的简化，程序运行过程中有一些数据不完整。

从下一节开始介绍 DSP/BIOS 各个组件的大体情况及其 API 函数。

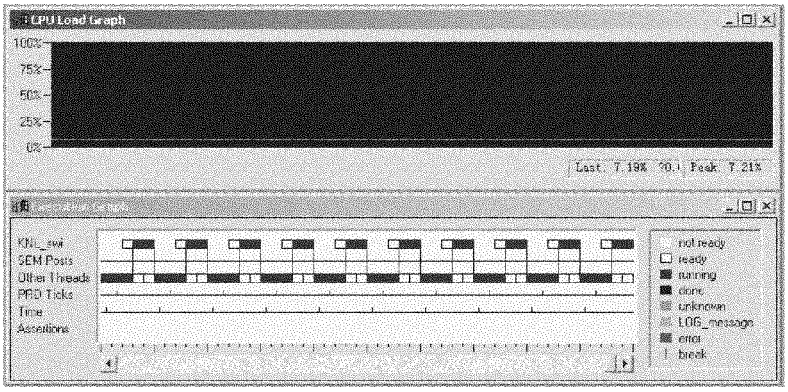


图 4-30　CPU 负载和线程运行图

4.3　DSP/BIOS 组件

本节按图 4-31 中列出的各个组件依次进行介绍，最后一个组件 CSL 即 Chip Support Library 将在下一节中集中介绍。

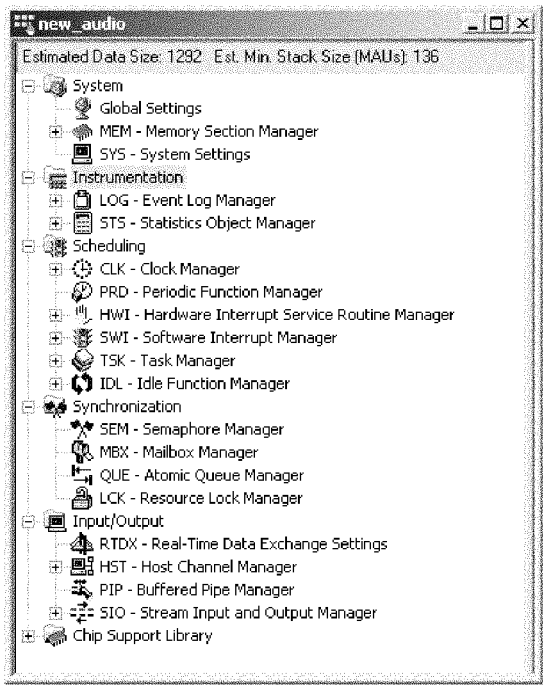


图 4-31　DSP/BIOS 组件管理器

4.3.1　System 栏

System 一栏即系统栏，包括下面三个项目：

1) Global Settings

这个项目设置的属性有下列几种：

Target Board Name：目标板的名称，往往根据所用 DSP 的类型自动命名。

DSP Speed In MHz (CLKOUT)：DSP 的工作时钟，一般地，CLKOUT 设为工作时其时钟与 DSP 的工作时钟相同。

DSP Type：DSP 芯片家族的类型，如 C54x。

Chip Support Library (CSL)：指定具体 DSP 芯片的类型，如 C5402。

Chip Support Library Name：指定 DSP 芯片的片上外设支持库。

PMST(6-0)：PMST 映射寄存器的低七位，即 MP/MC、OVLY、AVIS、DROM、CLKOFF、SMUL 和 SST，仅有这七位可以修改，各位具体的含义可以参考在线“帮助”。

PMST(15-0)：PMST 寄存器的值。

SWWSR：软件等待状态寄存器的值，它控制软件可编程等待状态发生器。BIOS_init 初始化时，SWWSR、BSCR 和 CLKMD 寄存器将被初始化，之后才调用 main 函数。

BSCR：直译为块开关控制寄存器的值。这个寄存器的功能在于防止访问不同存储区域切换时在总线上发生冲突。

Modify CLKMD：修改下面的设置。

CLKMD - (PLL) Clock Mode Register：CLKMD 寄存器的值，需要在 PLL 模式下调整 CPU 内部时应设置这个值。

Function Call Model：使用扩展模式函数编译连接方式。

Call User Init Function：修改下面的选项，指定一个初始化函数，可以在 main 函数前，且在 BIOS 初始化之后调用，但是这个函数不能包括任何 API 函数。

User Init Function：指定的初始化函数名。

Enable Real Time Analysis：这是个优化选项，如果不选中，可以去掉一些 DSP/BIOS 的实时分析工具和 LOG、STS、TRC 等模块的 API 函数。

Enable All TRC Trace Event Classes：启动 TRC 实时跟踪事件。

2) MEM(Memory Section Manager)

这是存储区段管理器。在这个项目中用户可以图形化地分配自己的 DSP/BIOS 程序存储区段的占用情况，DSP/BIOS 默认的各存储区段均已列出，用户可以直接通过右键单击后的弹出菜单进行修改。这一部分与前面所讲的存储器分配相类似，不再冗述。

3) SYS-System Settings

这个项目包括的属性有下列几种：

Trace Buffer Size：定义系统信息的跟踪缓冲区。

Trace Buffer Memory：指定这个缓冲区所在的存储区段。

Abort Function：当调用 SYS_abort 函数时，会执行设定在此处的函数，缺省为 _UTL_doAbort。如果设定的函数用 C 语言编写，此处函数设定名前应加上一个下划线，程序中的 C 语言函数名是没有这个下划线的，这个性质对于下面的属性也适用。

Error Function：当调用 SYS_error 时，设置在此处的函数被执行，缺省为_UTL_doError。

Exit Function：当调用 SYS_exit 时，会执行设置在此处的函数，缺省为 UTL_halt。

Putc Function：设置在此处的函数当遇到调用 SYS_putchar、SYS_printf、SYS_vprintf 时被执行，缺省为_UTL_doPutc。

4.3.2　Instrumentation 栏

这一栏可以称为“示波器”栏，包括两个项目：LOG 和 STS。

(1) LOG 组件可以建立 LOG 模块，这个模块可以调用 LOG 组件的 API 函数捕获实时信息，在 CCStudio 给出的 hello 程序段就使用了这种方法。

(2) STS 组件中建立的对象有一个重要属性 unit type，通过这个属性可以设定在 STS “示波器”中显示的时间轴的时间单位。在 C6000 中更为方便一些，可以指定时间单位。

4.3.3　Sheduling 栏

这一栏可以称为“课程表”，包括了六个组件：CLK、PRD、HWI、SWI、TSK 和 IDL。

(1) CLK 组件：这是一个具有重要用途的组件，通过它定义的对象可以指定一个被计时器中断调用的函数。几乎所有的硬件中断能调用的任何 DSP/BIOS 操作它都可以调用（HWI_enter 和 HWI_exit 除外）。这个被时钟调用的中断服务程序应尽量短小，因为经常被调用，要避免占用太多的 CPU。

(2) PRD 组件：与 CLK 组件相类似。

(3) HWI 组件：前面多次用到该组件，它负责管理 DSP 的所有硬中断源，这些中断源名是不可更改的，当使用 C 语言编写中断服务程序时，一定要打开中断通知（Use Dispatcher，直译为中断发射器），DSP/BIOS 会自动管理中断调用时的环境保护。

(4) SWI 组件：前面已介绍过该组件，其工作方式与 HWI 相同，它是由 API 函数来触发的。

(5) TSK 组件：可以用 TSK 组件构建 TSK 对象执行相关的任务或其 API 函数。当执行这些任务时可以设定一些函数被执行。

(6) IDL 组件：该组件也可以被看作是另一种意义下的 TSK 组件，它们的工作原理是相似的。

4.3.4　Synchronization 栏

这一栏可以称为同步信息栏，包括四个组件，都可以定义相应的对象并调用相应的 API 函数。

4.3.5　Input/Output 栏

Input/Ouptut(输入/输出)栏包括四个组件：RTDX、HST、PIP 和 SIO。PIP 组件的应用前面已介绍过，后面第五章将介绍 RTDX 技术，HST 组件与 RTDX 有关，负责主机的数据通道管理。

4.3.6 API 函数

DSP/BIOS 提供了一个灵活的用于图形化编程和实时调试的编程内核，以其高度模块化、面向中断的编程方法，节约了 CPU 的占用时间，为用户进行 DSP 应用系统设计带来了极大的方便。实现 DSP/BIOS 的功能是通过调用其相应的 API 函数来完成的。DSP/BIOS 提供了 150 多个 API 函数，分别针对不同的组件对象，可以被 C/C++和汇编语言直接调用（对于 C 语言，有些只能间接调用，而且有时考虑到为了提高调用速度时，常用汇编语言编写中断入口程序）。

API 函数分别属于不同的 DSP/BIOS 组件对象，所以相应的 API 函数也分为几个程序包（或称函数包），依次为：

(1) std.h 和 stdlib.h。所有的 DSP/BIOS 程序都应将 std.h 作为第一个包括文件，它包含了 DSP/BIOS 里的一些专用声明，stdlib.h 为通用的 C 库函数，这在前面有详细的介绍。

(2) ATM 程序包。该程序包包括了一些用汇编语言写成的原子功能（或称微功能）API 函数，直接面向硬件，供其他 API 函数调用。

(3) C54 或 C55 程序包。这是面向 C54 或 C55 的专用程序包，这两个包互不兼容。所以说，虽然 DSP/BIOS 提供了更高层次的硬件抽象，为平台间的移植带来了方便，但这个移植主要是指在一个系列中的移植，例如 C54xx 之间的移植，C55xx 与 C54xx 是两个不同的系列，它们之间一般不能直接移植，但不是不可移植，需要适当做些修改。

(4) 下面的一些是针对相应的组件进行命名的，如 GBL、MEM、SYS、LOG、STS、CLK、PRD、HWI、SWI、TSK、IDL、SEM、MBX、QUE、LCK、RTDX、HST、PIP、SIO 等。以上按 DSP/BIOS 组件管理器中的顺序排列，相应的 API 程序包中的函数以这些组名加下划线开头（也有极少例外）。在使用相应的那个组件时可以自行去参考在线帮助文件。

(5) 此外，还有一个 DEV 程序包，负责 DSP 各种片上资源的驱动；还有一个 TRC 程序包，包括一些实时信息采集和跟踪函数。

这些 API 函数是 DSP/BIOS 的真正灵魂，其调用格式与普通 C 函数相似，读者可以根据需要去查阅相应的 API 程序包，限于篇幅，这里不再举例。

4.4 CSL 组 件

CSL 组件是 DSP/BIOS 中的一员，它的主要作用在于为用户提供 DSP 片上外设的抽象调用方法。CSL 通过片上外设的硬件抽象，使用户不用去管理具体硬件的细节（包括时序）而直接引用即可。这不但方便了用户开发硬件系统，而且提供了一种硬件访问的标准化方法，使程序具有更好的可移植性和灵活性。另外，在加快了开发速度的同时，也减少了访问出错的机会。CSL 支持调用 DSP 片上所有外设，对于 C54 来讲包括 DMA、GPIO（通用 I/O 口）、MCBSP、PLL、TIMER 和 WDTIMER（看门狗计时器）等。

CSL 与 DSP/BIOS 一样，是通过专用的 API 函数使得访问片上外设更加方便。这些 API 函数按功能不同分为许多个程序包，依次为：CHIP 程序包、DAT 程序包、DMA 程序包、EBUS 程序包、GPIO 程序包、HPI 程序包、IRQ 程序包、MCBSP 程序包、PLL 程序包、PWR

程序包、TIMER 程序包和 WDTIM 程序包，分别支持和管理 CPU、DMA 数据搬移、总线、通用 IO 口线、HPI、片上外设中断、McBSP 口、PLL、掉电工作模式、计时器等片上资源。各个程序的 API 函数大部分以相应的程序包名加下划线开头，容易识别和掌握。

在程序中调用 CSL 的方法有两种：一种是通过 DSP/BIOS 组件管理器，另一种是直接调用 CSL 库函数。后者要求用户对 CSL 的各种库函数相当了解，这种方法生成的可执行程序代码要小得多。但是建议读者使用第一种方法，即借用 DSP/BIOS 组件管理器，特别是当用户使用中断时，使用 DSP/BIOS 更为方便一些。

几乎任何一个程序都离不开 CSL 的支持，因为任何一个程序的完成必然是对数据的处理，这些数据源必然是来自片上外设的，除了访问片上外设之外，没有其他的途径。

4.5　本 章 小 结

本章介绍了一种 DSP/BIOS 编程技术，通过这一章的学习，读者应明白 DSP/BIOS 的作用和 CSL 的意义，知道何时程序中可以使用 DSP/BIOS 技术，懂得怎样使用 DSP/BIOS 的七个实时软“示波器”对自己的程序作出各种指标评价。

通过这一章的学习，读者应大体了解 DSP/BIOS 和 CSL 的各个 API 程序包及其调用方法，应会查找相应的 API 函数，充分了解 DSP/BIOS 的各个组件的含义和应用。

更重要的是应充分理解 DSP/BIOS 程序的工作原理，学会 DSP/BIOS 编程。应强调指出，DSP/BIOS 程序是依次调用 BIOS_init、main、BIOS_start 各个函数，最后进入到空闲状态。空闲状态并不是什么都不做，而是一个空闲的进程不断重复，用户可以设计自己的 main 函数，其他的程序均由 DSP/BIOS 自动生成和管理（自动调用），不需要用户显式声明调用，用户需要完成的数据处理任务都集中在中断程序中完成。一句话，DSP/BIOS 帮助完成了各个进程间的管理，主函数 main 的作用被减弱了，甚至可以为空函数（注意：不能不要）。

习　题　四

1. 什么是 DSP/BIOS？
2. DSP/BIOS 的主要用途是什么？
3. 如何使用 DSP/BIOS 组件管理器？
4. 如何设置 DSP/BIOS 使其中的动态调试信息不被包括在可执行目标文件中？
5. 什么是 DSP 的软中断和硬中断，它们有什么区别？
6. 编写一个小的 DSP/BIOS 程序，实现串口数据的交换。
7. 编写一个小的 DSP/BIOS 程序，实现 DMA 的功能。
8. DSP/BIOS 程序的 MAP 表为什么会很大？
9. 如何生成 DSP/BIOS 程序的 MAP 表？
10. 如何调用 API 函数？
11. 简述 DSP/BIOS 的组件及其特性？
12. DSP/BIOS 是如何工作的？主函数起什么作用？
13. DSP/BIOS 程序用不用显式调用初始化函数？

14．第 4.2.2 节的程序有何缺点？为什么？应怎样改进？

15．如何进行 DSP/BIOS 中断编程？

16．简述第 4.2.4.1 节程序的工作原理。

17．第 4.2.4 节中简化后的程序与仿写的程序有何不同？各适用于什么情况下？

18．什么叫 DSP/BIOS 的软“示波器”？怎么使用？

19．什么是 CSL 组件？调用这个组件的编程方法有哪两种？

第五章

RTDX 程序设计

5.1 本章内容简介

本章介绍一种实时数据交换技术和建立在这个技术基础上的程序设计，即 RTDX 程序设计。RTDX 是 TI 公司的一个注册商标，是 TI 公司为实现计算机与 DSP 芯片之间的实时数据交换提出来的。可以将 RTDX 理解为一个时分复用的全双工数据通信管道，在这个过程中，计算机被称为主机，DSP 芯片被称为目标机，RTDX 现在更多的是被理解为主机与目标机之间进行实时通信的一种协议名称。

基于 JTAG 接口实现的 RTDX 的传输速率是很低的，只有几十 kHz 的采样率。现在的 XDS560 系列仿真器采用高速 RTDX 技术，可以实现视频信号的在线测试。对本章的 RTDX 的理解不应放在通信的角度上，而应放在测试实时数据的角度上；RTDX 可以实现主机与目标机之间的数据传输而不占用 DSP 的有用资源，所以是一种辅助性的程序数据校验技术。

RTDX 在主机和目标机之间传送数据也像是流过两个水库的一条河道，数据会在一个水库中缓冲一下，然后流到下一水库缓冲一下，视数据传输的方向，两个水库分别位于目标机和主机上，在一个时刻水流只能有一种流向。作者觉得数据的输入/输出和著名数据库软件 PowerBuilder 有点相似，对于不同系列的 DSP 芯片，方法略有不同，原理完全一样。

值得一提的是，一些著名的信号处理软件包，如 MATLAB 和 SystemView 等，都集成了 RTDX 的接口，当然这些软件也提供了与 CCStudio 之间的工程文件快速原型的技术，在第三章中提到的滤波器，就是直接由 MATLAB 生成到 CCStudio 中去的。微软公司的程序设计软件包可以直接访问 CCStudio 的 RTDX 数据管道，这种访问是建立在 OLE(对象链接与嵌入)技术上的，所以也可以用作主机端数据处理软件。RTDX 也支持多 DSP 与主机的数据交换。计算机模拟环境下也是支持 RTDX 的，虽然这本身有点儿画蛇添足的感觉，但是可以略加修改用于仿真环境下。本章也将介绍如何进入计算机模拟工作环境，并使得模拟环境与仿真环境并存。同前面几章一样，本章也将介绍基于 SY-5402EVM 板仿真环境下的主机和目标机双向数据传输程序的设计方法，介绍 Visual Basic 6 和 MATLAB 6 作为主机服务软件包的程序的设计方法（本章给出了完整的实例），并以实例进一步介绍 RTDX 技术。此外，基于计算机模拟环境下的 RTDX 程序运行速度较仿真环境下慢一些，RTDX 可以嵌入到 DSP/BIOS 程序中。

思考题

(1) RTDX 的本质作用是什么？
(2) 为什么装入.out 程序到目标板上时要关闭 RTDX（即不能开启或“使能”RTDX）？
(3) MATLAB 下实现 RTDX 的编程设计的原则是什么？
(4) SystemView 下实现 RTDX 的设计方法是什么？

5.2　计算机模拟环境设置

本章主要介绍建立在 SY-5402EVM 板的仿真环境下的 RTDX 的程序设计。计算机模拟环境下支持的 RTDX 程序设计没有实质性的用途，仅是为了方便那些在现场没有仿真环境的用户能在计算机模拟环境下进行 RTDX 程序设计，但是这些程序略加修改即可应用到仿真环境下。本节介绍如何使 CCStudio 工作在模拟环境下，并借此介绍什么是 CCStudio 的 PDM 管理器。

？ 第一步：点击桌面上的 Setup CCS2 图标，进入图 5-1 所示的窗口。

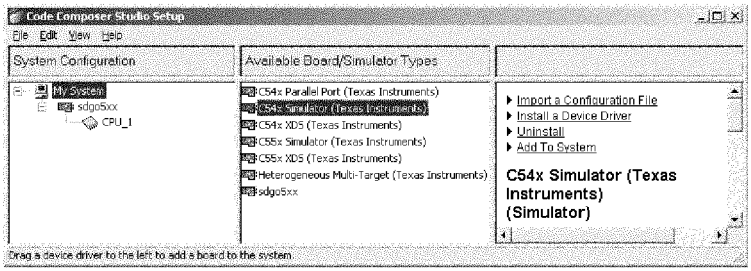


图 5-1　CCStudio 配置目标板窗口

在图 5-1 中，左边是前面几章使用的驱动程序所建立的目标板配置，这个可以保留不变，CCStudio 支持多目标板工作。将图 5-1 中间一栏 C54x Simulator(Texas Instruments)拖拉到最左边的一栏内，出现图 5-2 所示窗口。

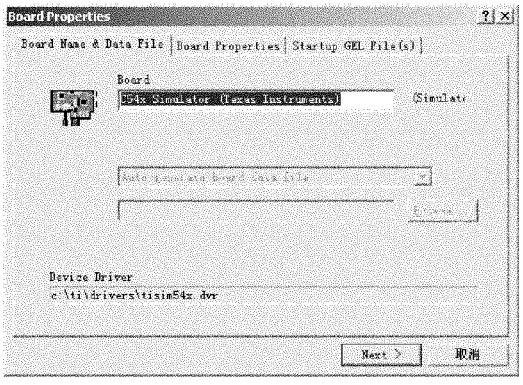


图 5-2　指定目标板

图 5-2 所示窗口不用更改，直接点击 next 进入图 5-3 所示窗口。在这个窗口中，可以根据自己的需要设置相应的模拟 DSP 芯片文件。图 5-3 中选择了 SIM5402.cfg 文件，点击 next 进入图 5-4 所示窗口。在此，要相应地设置成同一个芯片的 GEL 文件，这里可以空着不填入，等进入到 CCStudio 下，再加入一个 GEL 文件。在模拟环境下，加上一个 GEL 文件是必要的，它将初始化整个模拟环境。

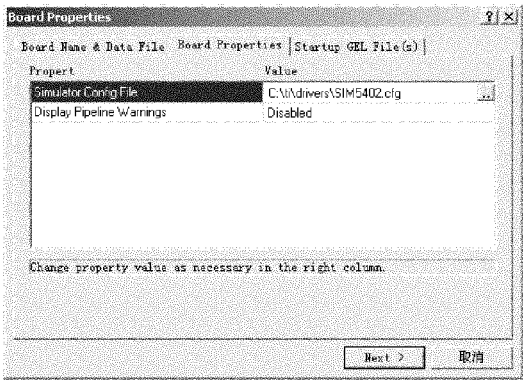


图 5-3　指定配置文件

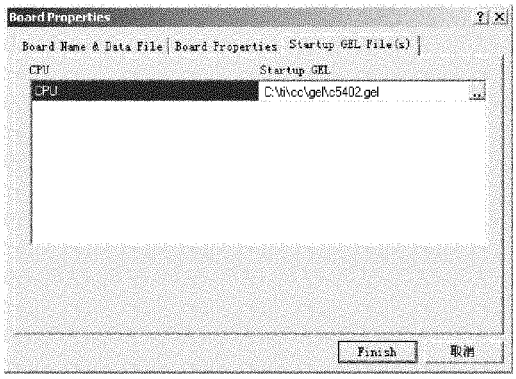


图 5-4　设置 GEL 文件

？ 第二步：在图 5-4 中点击 Finish 进入图 5-5 所示窗口。在图 5-5 中 My System 下将多出一个刚刚配置的目标板，这个目标板是用于计算机模拟环境下的，这一章的 RTDX 程序调试运行也可以借助这个环境。

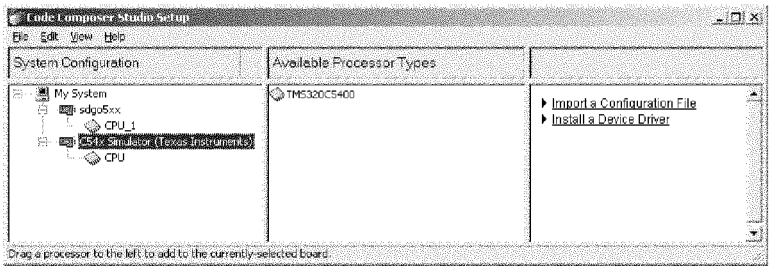


图 5-5　CCStudio 配置完成的窗口

？ 第三步：保存这个配置，退出 Code Composer Studio Setup，并进入 CCStudio 工作环境，将出现图 5-6 所示的窗口，这个窗口称为并行调试管理器。点击图 5-6 中的 Open 菜单，得图 5-7 所示画面。在图 5-7 的 Open 菜单中点击子菜单项：C54x Simulator(Texas Instruments)/CPU，即可进入模拟工作环境，如图 5-8 所示。注意图 5-8 窗口中的标题栏，前面的仿真模式工作窗口如图 1-9 所示，其标题栏为：

sdgo5xx/CPU_1

现在是：

C54x Simulator(Texas Instruments)/CPU

图 5-6　并行调试管理器

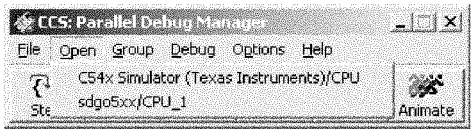


图 5-7　并行调试器（PDM）的 Open 菜单

从图 5-7 中可以看出，CCStudio 可以同时仿真或模拟多个 DSP 功能板，本书不再细讲。图 5-8 就是 RTDX 工作的模拟平台，这里选用了 VC5402 的模拟环境进行介绍，其实在同系列中选用什么样的 DSP 芯片进行模拟并不关键。图 5-7 和图 5-8 在多 DSP 仿真环境或是模拟环境下是同时存在的，用户可随时切换到 PDM 下的其他环境下工作，对当前的环境没有任何影响。所以说可以在图 5-7 中打开仿真环境，同时调试运行一个仿真程序；也可以在如图 5-8 所示的模拟环境下运行 RTDX 程序，两者可兼得并存。

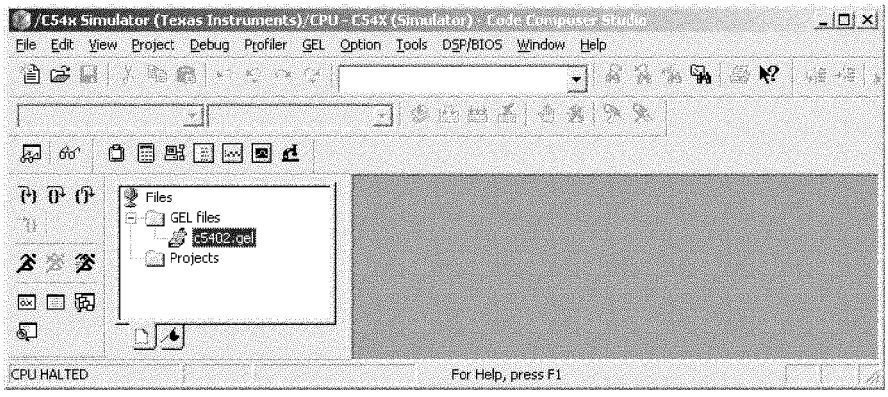


图 5-8　计算机模拟工作环境（注意看窗口标题）

5.3　RTDX 编程基础

5.3.1　RTDX 的数据交换协议

RTDX 本身代表了一种数据交换协议，如图 5-9 所示。

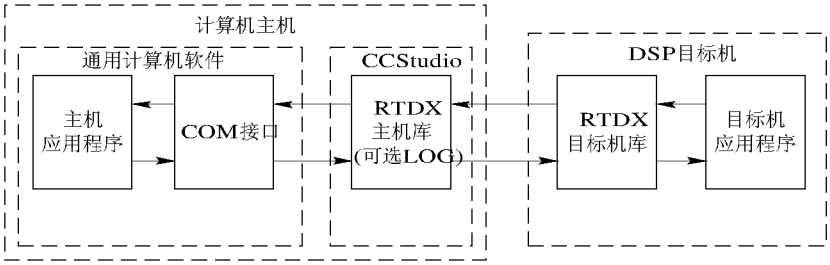


图 5-9　RTDX 数据通信框图

仅看图 5-9 的两端，可见 RTDX 最终实现了主机应用程序和目标机应用程序之间的数据通信，这是 RTDX 的本质。这个过程的完成需要借助 RTDX 分布在主机和目标机的支撑库以及主机的 COM（通用组件对象模型）接口。可选的 LOG 文件也可以将从目标机来的数据保存起来。

5.3.2　RTDX 配置

进入 CCStudio 仿真环境下，点开 Tools 菜单下的 RTDX 子菜单中的三项，会弹出如图 5-10 所示的三个窗口。

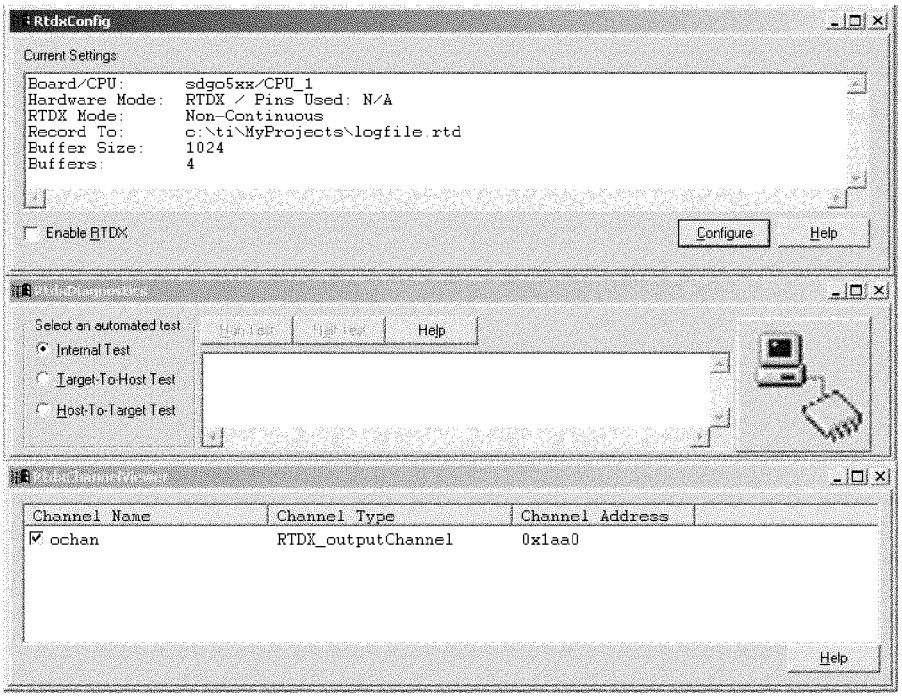


图 5-10　RTDX 的专用工具窗口

图 5-10 的 RtdxChannelViewer 窗口为 RTDX 程序中所用到的数据通道的情况。这个窗口可以手工加入数据通道，一般是由程序来指定，并定义为输出或输入数据通道。从这个

窗口中可以看到通道的类型和地址，通道名是可以随意指定的合法标示符。这个窗口信息可供用户编写主机程序时参考。

图 5-10 的 RtdxDiagnotics 窗口为通道测试窗口。这个窗口提供了测试主机与目标机之间数据通道的连接情况，Internal Test 为计算机内部 Simulator 的测试，下面两个测试项依次为目标机到主机和主机到目标机的测试。用户在使用这个测试程序时，不一定能测试成功，原因可能有两个：其一是在图 5-10 的 RtdxConfig 图中没有启动 RTDX；其二，就是可能已经启动了 RTDX，但是仍然测试失败。这是因为 RTDX 要求在编译连接和装入目标程序.out 时应将 RTDX 关闭，即不能启动 RTDX（特别是在写入时，数据流经 JTAG 口线，可能会占用 RTDX 的数据通道，使 RTDX 通道初始化失败），而测试时要求先启动 RTDX，但是测试的第一步是将程序写入到目标板上去，显然，这是一对矛盾。所以，有时测试失败（主要是测试超时）是正常现象。解决这个矛盾的方法是先写入测试程序，再启动 RTDX，然后运行程序进行测试，不要去点击 RtdxDiagnotics 窗口的 Run Test 按钮进行测试（测试程序读者可在在线“帮助”中找到）。

图 5-10 的 RtdxConfig 窗口是 RTDX 最重要的一个窗口，它完成对 RTDX 的配置。在不启动 RTDX 时，点击 Configure 进入图 5-11 和图 5-12 所示的窗口。

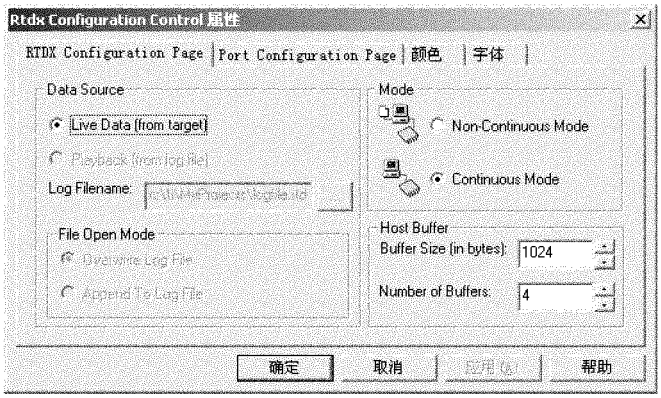


图 5-11　RTDX 配置页面

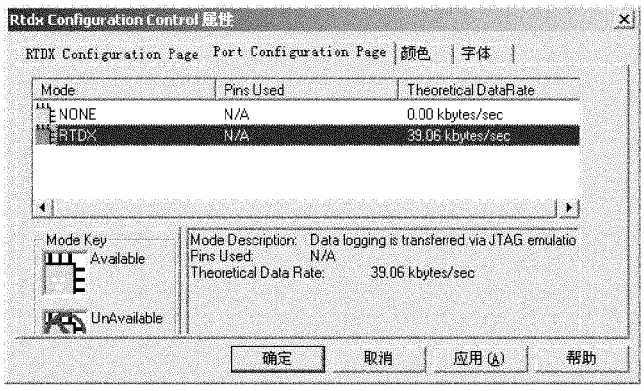


图 5-12　RTDX 端口配置页面

图 5-11 中 Mode 栏里有 Non-Continuous Mode 和 Continuous Mode 两个项目。前者指主机将接收到的来自目标机的数据存入一个 Log 文件中（即数据日志文件），后者是将数据存入循环式缓冲存储区域中。一般来说，图 5-10 的 RtdxConfig 窗口中的内容是不需要修改的，开机时默认不启动 RTDX。

5.3.3　RTDX 程序设计流程

RTDX 可以在不干扰目标机上程序正常运行的情况下，实现主机与目标机之间的数据交换(双方主要是交换数据)。对于目标机向主机传送数据来说，不但可以传送数据，还可以传送标志事件运行状态的数据，这些数据常被称为事件数据，用户在传送事件数据时要定义这个事件数据才行。主机向目标机则主要是传送各种类型的普通数据。

无论是目标机向主机传送数据，还是主机发送数据到目标机，都可以分为以下几步来完成：

? 第一步：编制目标机（DSP）程序用来发送或接收数据。这一步是在 CCStudio 下完成的。这一步必须先行，因为在这一步中需要定义用于主机访问的 RTDX 数据通道并向这些通道写入或读取数据，RTDX 编程有无必要，关键在于看这一步有没有必要。

? 第二步：编写主机（计算机）程序用于接收或发送数据。这一步是在 Visual Basic 或 MATLAB 等软件下完成的。这一步需要使用第一步中定义的 RTDX 数据通道，并通过这些通道读取或写入数据。

以上这两步的程序设计格式对于同一系列的 DSP 芯片来讲是固定的，对于不同系列的 DSP，其格式稍有不同。

? 第三步：将目标程序写入到目标机上。特别注意，因为 RTDX 是通过 JTAG 完成数据交换的，在编译连接和写入程序时，一定不要启动 RTDX，否则程序运行是没有结果的。

? 第四步：启动 RTDX。

? 第五步：运行目标机程序。

? 第六步：运行主机程序。

? 第七步：在主机上对目标机传送来的数据进行分析（本书中没有涉及，这是通用数字信号处理算法）。

根据上述几个步骤，具有 SY-5402EVM 板的读者可以将计算机的音频输入端接入到 SY-5402EVM 板上，通过 RTDX 再将数据读入到计算机中去，将这些数据整理成一个 .wav 文件，用计算机重放音乐或者观测这些数据的频谱。

5.4　使用 Visual Basic 的 RTDX 程序设计

5.4.1　程序功能介绍

本程序在 SY-5402EVM 板上调试通过，本程序完成的功能是通过 Visual Basic 编写主机应用程序，由主机应用程序发送一个整型数组到目标机上，然后目标机将这些整型数据加倍后回送到主机应用程序中。

5.4.2 目标机程序设计

RTDX 程序设计分为目标机程序设计和主机程序设计两部分。一般地，目标机程序设计是先进行的，在目标机程序设计中需要定义 RTDX 的数据通道，这些通道将在主机程序中引用。所以说，通道的名称是相同的，但是具体的名称由用户习惯指定。在本示例程序中，定义输入通道为 inchan，输出通道为 ochan。

RTDX 的程序设计几乎没有任何技巧，而是遵循固定的格式。对于目标机的程序设计来说，可以分为以下几个步骤：

? 第一步：编写主程序。这是用户仅需要做的一件事情，其他步骤的文件可以选用 CCStudio 提供的文件。主程序文件必须包括 rtdx.h 头文件，这个头文件中定义了 RTDX 的 API 函数；主程序中必须创建全局的 RTDX 输入和输出通道，通道名称可以是任意合法的标示符；在主函数中将这些通道启动，通道必须先启动才能被使用；进行通道的读入和写出操作，对于 C5000 来说，读入和写出形式上有点不同，但对于 C6000 来说，形式上是相同的。在 C5000 中，向输出通道写入数据时，必须加一个 RTDX_Poll()函数对数据输出作一个确认，而读数据时，这个函数被隐式执行了；RTDX 读写有专用的 API 函数，只需调用即可；RTDX 数据交换完成后，必须将通道关闭。

? 第二步：建立一个新的工程文件 user_rtdx.pjt，向这个工程文件中加入刚才编写的主程序 user_rtdx.c，同时加入 rtdx.lib、rts.lib、vectors.asm 和 user_rtdx.cmd 等文件。库文件是由 CCStudio 提供的。

注意：RTDX 对于仿真环境和模拟环境下的支撑库是不一样的。

? 第三步：中断向量表文件和存储器配置文件可以自己编写，也可以直接拷贝相应目标板下 shared 目录里的 intvecs.asm 和 c5402dsk.cmd（对于本实例），特别注意自己编写中断向量表时，应加入对 RTDX 中断屏蔽字的宏定义。

? 第四步：编译连接成可执行文件。在这个过程中一定要关闭 RTDX。

? 第五步：把可执行文件写入到目标板上。在这一步中一定要关闭 RTDX。写入完成后，才能打开 RTDX。

可见，一般只有运行程序时打开 RTDX，程序运行完后应立即关闭，因为它占用 JTAG 口线。

在第 5.4.4 节，仅给出了主程序的源代码，其他程序都是 CCStudio 自带的，没有给出。

5.4.3 主机程序设计

主机程序设计可以在很多软件平台上进行，如 Visual Basic、Visual C++、VBA(Access)、MATLAB 和 SystemView 等等。这一节中我们将集中介绍 Visual Basic（以下简称 VB）下的主机程序设计方法，在下一节中给出了这个程序的源代码。具体设计步骤如下：

? 第一步：进入 VB6 编程环境，新建一个标准工程文件 project1.vbp，这个工程文件中缺省只有一个窗体 Form1，对本程序来说，这个 Form1 已经足够了。将 Form1 的 Caption 属性更名为“RTDX 主机应用程序”，向这个窗体中依次加入两个 Label 标签、两个 List 列

表框和两个 Command 命令按钮，将它们的标题 Caption 属性依次更名为“主机向目标机发送的数据”、“主机接收目标机的数据”、“主机发送”和“主机接收”等，如图 5-13 所示。

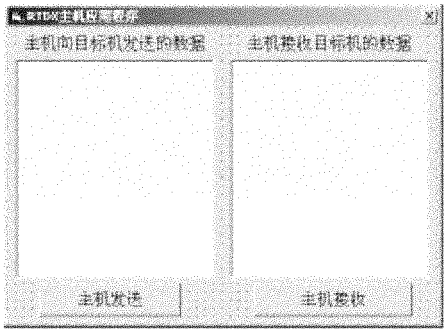


图 5-13　主机程序界面

? 第二步：点击 View Code 图标进入代码编辑窗口，编写全部的程序代码。在编写 RTDX 主机端程序时，也是有固定格式的：首先是定义并创建一个 RTDX 全局对象，进一步打开这个 RTDX 对象，打开对象时要指定它与目标机的读入通道还是写出通道相连接；然后进行读入或写出数据的操作，有专用的函数可以供调用；最后，完成读入和写出数据操作后，关闭 RTDX 对象。

? 第三步：选择 File 菜单下的生成可执行文件子菜单，生成一个可以直接在 Windows 环境下运行的可执行程序，程序名可以自己指定，这里指定为 Host2Targ.exe。

? 第四步：在 Windows 桌面上创建一个快捷方式程序图标，使这个程序图标指向 Host2Targ.exe，可以方便用户使用。

5.4.4　程序运行结果及源代码

因为程序格式固定、结构清晰，这里就没有给出具体的程序流程框图。程序运行过程是：首先，在 CCStudio 环境下将 .out 可执行文件写入到目标机中，即将程序下载到 DSP 芯片中；其次，点击桌面上的主机程序图标，运行 Host2Targ.exe 主机程序，这时切记不要去点击“主机发送”和“主机接收”两个命令按钮；再次，启动 RTDX，并运行目标机上的程序，目标机上的程序运行时会创建 RTDX 数据通道，供主机调用；最后，点击主机程序界面中的“主机发送”命令按钮，可以在列表框中看到主机发送数据的情况，然后再点击“主机接收”命令按钮，可以在相应的列表框中看到主机发送到目标机的数据被倍乘后送回来。图 5-14 为目标机程序运行时动态创建的 RTDX 数据通道及通道属性。主机程序的运行结果如图 5-15 所示，同时目标机的情况在 CCStudio 的标准输出窗口中显示出来（这是编程指定的），如图 5-16 所示。

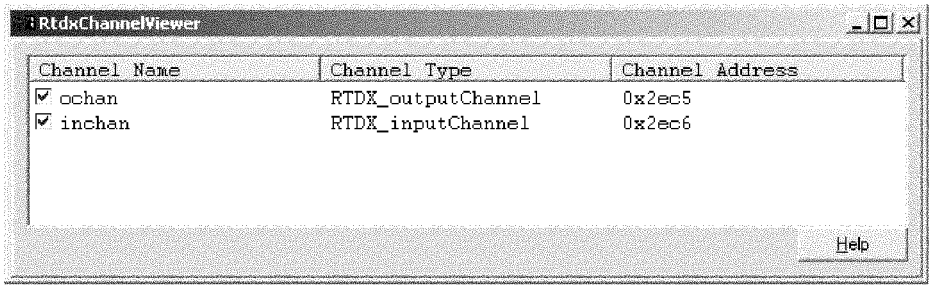


图 5-14　RTDX 的数据通道

图 5-15　主机程序运行结果

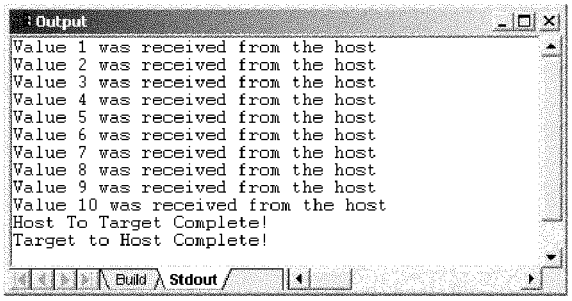


图 5-16　目标机接收和发送数据的情况

下面首先给出目标工程文件中的主程序文件内容，其他的文件都可以在 CCStudio 中原程序调用或引用。

// 目标机的主程序文件 user_rtdx.c

```
/*filename: user_rtdx.c*/

#include <rtdx.h>          /* defines RTDX target API calls          */
#include "target.h"        /* defines TARGET_INITIALIZE()            */
#include <stdio.h>         /* C_I/O                                  */

RTDX_CreateOutputChannel( ochan );      /* Must declare a global output channel */
                                        //定义一个全局的输出通道
RTDX_CreateInputChannel( inchan );      /* Must declare a global input channel */
                                        //定义一个全局的输入通道
void main()
{
    long h2tdata[10];
```

```
long status,i;

int t2hdata[10];

TARGET_INITIALIZE();                    //目标板的初始化
RTDX_enableOutput( &ochan );            /* Enable the output channel -ochan for t2h */
//上面为输出通道启动，下面为输入通道启动
RTDX_enableInput( &inchan ); /* Enable the input channel -inchan for    h2t */
/*Host To Target Transmits An Array of Integers*/
status = RTDX_read( &inchan, h2tdata, sizeof(h2tdata) );        //目标机接收主机数据
if ( status != sizeof(h2tdata) )
{
        printf( "ERROR: RTDX_read failed!\n" );
        exit( -1 );
}
else
{
        for( i = 0; i < (sizeof(h2tdata) / sizeof(long) ); i++)
        {
                printf( "Value %ld was received from the host\n",h2tdata[i] );
                t2hdata[i]=2*h2tdata[i];
        }
}

                                        //关闭输入通道
RTDX_disableInput(&inchan);             /* disable the input channel*/

printf("Host To Target Complete!\n");

/*Target To Host Data Transmit an array of integer*/
status = RTDX_write( &ochan, t2hdata, sizeof(t2hdata) );
//目标机向主机发送数据
if ( status == 0 )
{
        puts( "ERROR: RTDX_write failed!\n" );
        exit( -1 );
}

while ( RTDX_writing != NULL )
```

```
    {
        #if RTDX_POLLING_IMPLEMENTATION          //已被宏定义为 1
            RTDX_Poll();                         //目标机发送数据确认
        #endif
    }
                                                 //关闭输出通道
    RTDX_disableOutput( &ochan );/* disable the output channel */
    puts( "Target to Host Complete!\n" );

}
//程序中的中文注解是后加入的
```

下面的程序段为主机应用程序。这个主程序的界面是在 VB 环境下画出来的，是典型的“所建即所得”的可视化界面设计。要调试这个程序时应在窗体中先画出这个界面，然后将程序段依次写入相应的事件例程中。

```
'全局声明
Option Explicit
Const Success = &H0 'Method call is valid
Const Failure = &H80004005 'Method call failed
Const ENoDataAvailable = &H8003001E
'No data was available. However, more data may be available in the future.
Const EEndOfLogFile = &H80030002
'No data was available.The end of the log file has been reached.
Dim rtdx As Object                    '定义一个 RTDX 对象
Dim status As Long
Dim i As Long

'自定义函数 DataReceive，用于主机接收 RTDX 数据
Sub DataReceive()

Dim vardata As Variant

On Error GoTo Error_Handler           '出错处理语句
    '创建主机数据接收通道
    Set rtdx = CreateObject("RTDX") 'Create an instance of the RTDX COM object
    status = rtdx.Open("ochan", "R") 'Open channel ochan for reading

    If status <> Success Then
        List2.AddItem ("Opening of channel ochan failed")
```

```
        GoTo Error_Handler
    End If

    Do
        status = rtdx.ReadSAI2(vardata)
        'Read a message (16 bit integer array) from the target.

        Select Case (status)
            Case Success

                For i = LBound(vardata) To UBound(vardata)
                'Print each element in the array
            List2.AddItem ("Value " & Str(vardata(i)) & " was received successfully")
                Next i
            Case ENoDataAvailable
                List2.AddItem ("No data is currently available")
            Case EEndOfLogFile
                'List2.AddItem ("End of log file has been detected")
            Case Failure
                List2.AddItem ("ReadSAI2 returned failure")
            Exit Do
            Case Else
                List2.AddItem ("Unknown return code")
            Exit Do
        End Select
    Loop Until status = EEndOfLogFile
    status = rtdx.Close() 'Close the channel                    '关闭接收通道
    Set rtdx = Nothing 'Release the reference to the RTDX COM object
    Exit Sub
Error_Handler:
    List2.AddItem ("Error in COM method call")
    Set rtdx = Nothing

End Sub

'单击命令按钮 Command1 的事件响应函数
Private Sub Command1_Click()

    Call DataTransmit
```

```
End Sub

'自定义函数 DataTransmit，用于主机向目标机发送数据
Sub DataTransmit()
Dim arraydata(10) As Long
Dim data As Long
Dim bufferstate As Long

On Error GoTo Error_Handler
    '创建发送数据通道
    Set rtdx = CreateObject("RTDX") 'Create an instance of the RTDX COM object
    status = rtdx.Open("inchan", "W") 'Open channel ichan for writing

    If status <> Success Then
        List1.AddItem ("Opening of channel ichan failed")
        GoTo Error_Handler
    End If

    data = 1
    For i = LBound(arraydata) To (UBound(arraydata) - 1)
    'Fill up a SAFEARRAY with values to send to the target
        arraydata(i) = data
        data = data + 1
    Next i

    status = rtdx.Write(CVar(arraydata), bufferstate)    '发送数据
    'Write a SAFEARRAY of 16-bit integers to the target

    If status = Success Then
        For i = LBound(arraydata) To (UBound(arraydata) - 1)
            List1.AddItem ("Value " & Str(arraydata(i)) & " was sent to the target")
        Next i
    Else
        List1.AddItem ("Write failed")
    End If


    status = rtdx.Close() 'Close the channel                '关闭通道
    Set rtdx = Nothing 'Release the RTDX COM object
    Exit Sub
```

```
Error_Handler:
    List1.AddItem ("Error in COM method call")
    Set rtdx = Nothing

End Sub

'单击命令按钮 Command2 的事件响应函数
Private Sub Command2_Click()
    Call DataReceive
End Sub
```

注：上面程序中的中文注释是后加入的。在 VB 中注释用“'”号开始，实际调试该程序时不要将中文写入。

5.5　使用 MATLAB 的 RTDX 程序设计

5.5.1　程序功能介绍

本程序在 SY-5402EVM 板上实现，完成的功能是由 MATLAB 编写主机应用程序。主机将发送一组整型数据到目标机上，目标机收到数据后，将这些数据乘三倍后回送给主机应用程序。MATLAB 应用程序与目标机应用程序之间的 RTDX 数据交换应用比较广泛，可以在 MATLAB 中将发送到目标机上的数据或是接收来自目标机的数据进行二次分析，以得出所需要的结论。

5.5.2　目标机程序设计

目标机程序设计与前面的第 5.4.2 节介绍的方法基本类似，只是将其中的一个语句：

```
for( i = 0; i < (sizeof(h2tdata) / sizeof(long) ); i++)
{
    printf( "Value %ld was received from the host\n",h2tdata[i] );
    t2hdata[i]=2*h2tdata[i];
}
```

由原来的放大二倍，变成本程序中的放大三倍，即为：

```
for( i = 0; i < (sizeof(h2tdata) / sizeof(long) ); i++)
{
    printf( "Value %ld was received from the host\n",h2tdata[i] );
    t2hdata[i]=3*h2tdata[i];
}
```

5.5.3　主机程序设计

主机程序也是在基于 MATLAB 的 GUI 界面进行设计的，分为以下几步完成：

? 第一步：新建一个 MATLAB 的 GUI 窗口，如图 5-17 所示。在这个窗口中放置一个坐标轴、两个命令按钮和三个静态文本框。其中，坐标轴用图形方式显示接收到的 RTDX 数据，一个静态文本框为图名，另外两个静态文本框用于动态显示执行程序的结果，两个命令按钮用于执行程序。

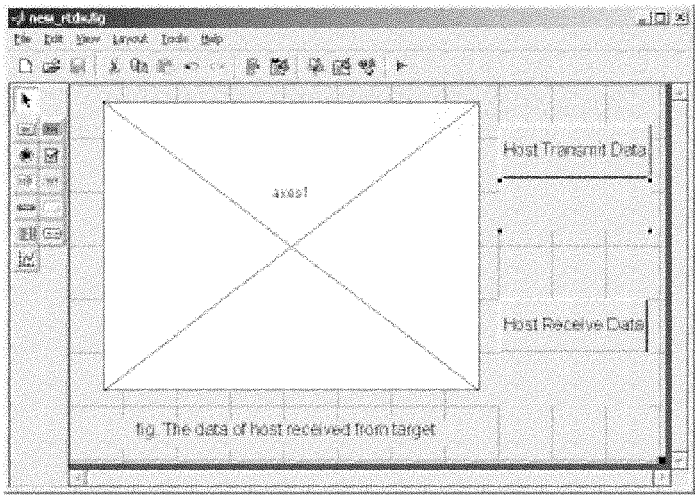


图 5-17　主程序界面

? 第二步： 使用 GUI 的方法建立用户界面，MATLAB 为这个界面自动生成一个同名的 .m 文件，进入到这个.m 文件中可以为两个按钮的功能编写相应的脚码程序。

? 第三步：在.m 文件的开头声明一个全局变量，用于访问界面中的各个对象。

? 第四步：在 Host Transmit Data 命令按钮下的回调函数中编写用于主机向目标机发送数据的程序，在 Host Receive Data 命令按钮的回调函数中编写用于主机接收目标机数据的程序。这些程序的编写有固定的格式，一般来说分为以下过程：其一，创建一个 RTDX 对象并将这个对象与相应的通道相连接；其二，对这个通道写入数据或从这个通道中读出数据；其三，关闭这个通道(或称释放这个通道)。对于这些操作, MATLAB 提供了相应的函数。

? 第五步：调试并执行这个 GUI 程序，这时不要去点击程序界面中的两个按钮。

5.5.4　程序运行结果及源代码

程序的运行严格按以下步骤进行：

? 第一步：关闭 RTDX，用 CCStudio 将目标机程序下载到 DSP 内部。

? 第二步：启动 RTDX，运行目标机程序。

? 第三步：运行主机程序，依次点击主机程序界面的两个按钮，可以得到如图 5-18 所示的结果。

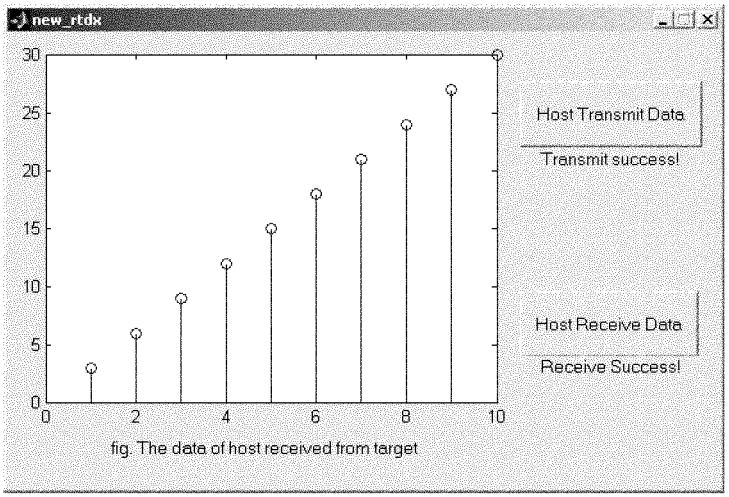


图 5-18　MATLAB 程序运行结果

图 5-19 是用 VB 程序测试的结果，放在这里是为了提供对比。图 5-18 用图形的方式表现了主机从目标机读取数据的结果。当然这个程序只是一个简易的版本，读者还可增加很多功能进一步扩充这个程序。VB 程序同样也有很大的可扩充空间。在图 5-18 中圆圈的中心对应的坐标为数据的值。当点击 Host Transmit Data 按钮，主机成功送数据到目标机后，这个按钮下面会出现“Transmit success!”字样。同样点击 Host Receive Data 按钮后，若主机成功地从目标机得到 RTDX 数据，则这个按钮下面会出现“Receive Success!”字样。

图 5-19　VB 运行结果

在 .m 文件中下面这一语句最后面应加入一个全局定义，如下：

```
function varargout = new_rtdx(varargin)
%加入的全局定义
global h_txt;
```

其他部分保持不变。需要补充的函数为下面的两个回调函数，分别执行发送和接收目标机数据的功能。全部程序代码列出如下：

```
%这一部分代码为主机发送数据到目标机
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
h_txt=handles.txt1;
cc=ccsdsp;
timeout_msg='Timeout';
nodata_msg='No more data is available';
errmsg=NaN;
%Host Transmit data to Target
indata=[1:10];
rtdx_ichan=cc.rtdx;                    %创建发送数据通道
rtdx_ichan.enable;
open(rtdx_ichan,'inchan','w');
while(isempty(findstr(timeout_msg,errmsg)))
    try   %read data from Rtdx_ochan
        writemsg(rtdx_ichan,'inchan',int32(indata));          %发送数据
        set(h_txt,'string','Transmit success!');
    catch
        errmsg=lasterr;
        %disp(nodata_msg);
        break;
    end
end
close(rtdx_ichan,'inchan');

%%这一部分代码为主机从目标机接收数据
function pushbutton2_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
h_txt=handles.txt2;
cc=ccsdsp;
timeout_msg='Timeout';
nodata_msg='No more data is available';
errmsg=NaN;
%Host Receive data from Target
```

```
rtdx_ochan=cc.rtdx;          %创建读取数据通道
rtdx_ochan.enable;
open(rtdx_ochan,'ochan','r');
while(isempty(findstr(timeout_msg,errmsg)))

    try   %read data from Rtdx_ochan
        data=readmsg(rtdx_ochan,'ochan','int16',[1 10]);    %读取数据
        stem(data);
            set(h_txt,'string','Receive Success!');
    catch
        errmsg=lasterr;
        %disp(nodata_msg);
        break;
    end
end
close(rtdx_ochan,'ochan');
```

5.6 本章小结

本章以两个具有代表性的实例介绍了 RTDX 的概念和应用，以便读者能掌握和使用 RTDX 程序的设计方法。

下面就本章开始提出的思考题作一简要的回答。

RTDX 的功能是实现主机应用程序与目标机应用程序之间的数据交换，并为这种交换定义相关的操作及数据交换协议。借用 XDS510 仿真器可能只有 39 kb/s 的速率，新的 XDS560 仿真器可以达到视频信号数据处理的要求。

向 DSP 芯片下载 RTDX 程序时，必须关闭 RTDX。这是为了解决一个简单的资源冲突问题。因为 JTAG 口的主要用途是作为仿真口，RTDX 也借用了这个口，为了避免下载时出现误码等错误，必须先关闭 RTDX。

RTDX 程序设计的原则与普通 DSP 的 C/C++程序设计的原则是相同的，追求高效性和实用性，面向应用。但是因为 RTDX 是用于主机和目标机的数据交换，对于主机程序设计来说，不但要追求数据处理的高效性，也要讲究程序设计的界面友好，只要速度足够快，主机应设计灵活方便的界面。

SystemView 是强大的专用通信仿真软件，这个软件提供了与 MATLAB 的接口。但它本身几乎是不需编程的，建什么模型就是什么通信实体的仿真，是真正意义的“所建即所得”的建模方法，现在已得到了广泛的应用，它也提供了对 RTDX 技术的完美支持。

习 题 五

1. 什么是 RTDX 技术？它的主要用途是什么？
2. 如何配置 RTDX？
3. 如何查看 RTDX 的数据通道的使用情况？
4. 设计 RTDX 程序的步骤是怎样的？
5. RTDX 通道由目标机定义还是由主机定义？为什么？
6. 主机应用程序一般使用什么软件来开发？
7. 使用 VB6 编写一个简单的 RTDX 程序，完成目标机向主机发送一个整数。
8. 使用 MATLAB6 编写一个简单的 RTDX 程序，完成主机向目标机发送一个整数。
9. RTDX 可以在主机和目标机之间交换什么样的数据？
10. 为什么 RTDX 测试有时不能成功？
11. 进行 RTDX 调试时需特别注意的事项是什么？什么时候才能开启或“使能”RTDX？
12. 对于 XDS510 系列仿真器，RTDX 的传输速率是多大？

第六章

Boot Loader 程序设计

6.1 本章内容简介

Boot Loader 是开发 DSP 应用系统必须做的最后一步工作。Boot Loader 方法是对单片机的一种改进。众所周知，通用单片机的程序是通过把单片机放入专用的烧写器中将程序烧入其中的 EEPROM 中，然后将单片机装入功能板上工作。DSP 为了增加软件下载的灵活性，将这个 EEPROM 等存储器放置到片外，由一片或几片 FLASH 来代替；DSP 的内部 ROM 固化了一个称为 Boot 的程序，在 DSP 上电硬复位后（MP/MC=0），DSP 自动执行这个 Boot 程序，将外部 FLASH 的程序读入 DSP 内部的高速 RAM 程序区中。所以，所谓的 Boot Loader 就是 DSP 上电后自动将固化在 FLASH 中的程序读入到 DSP 的片上 RAM 或片外 RAM 映射成的存储区间的一个过程。

CCStudio 生成的.out 可执行文件是 AT&T 的模块化 COFF 代码格式，这个格式因其具有模块化结构与实际的 FLASH 存储区间不匹配，所以不能直接写入到 DSP 内部或是 FLASH 上。CCStudio 提供了代码格式转化方法来完成这种匹配，也可以自己编写一个格式转换程序。

本章将就什么是 Boot Loader、实现 Boot Loader 的方法以及如何借助仿真器和 DSP 芯片将可执行程序写入到 FLASH 中的现场下载软件上（即现场 Boot Loader 或在线 Boot Loader）等进行详细的介绍，并给出了一个完整的基于 SY-5402EVM 板的在线 Boot Loader 程序实例。

所谓在线 Boot Loader 方法，是指通过仿真器和 JTAG 接口，在 CCStudio 软件平台上设计一个小程序，通过运行这个小程序，将 DSP 功能板上电后需要运行的程序写入到功能板的 FLASH 存储器内部（FLASH 是不易失的重复可读写存储器），必要时可以读出来进行校验。写入成功后，关闭 CCStudio、计算机、仿真器电源以及 DSP 功能板，将功能板与仿真器的连接断开。然后给 DSP 功能板单独上电，这时 DSP 内部的 Boot 程序会按外部中断或通用 I/O 口的设置，采用 ROM 中相应的 Boot 程序和 Boot 方法，从 DSP 功能板上的 FLASH 中读取程序，并将这些程序写入到 DSP 内部的高速 RAM 或片外映射到片上的外部 RAM（仅当内部 RAM 空间不够大时）。这个工作完成后，Boot 程序将程序指针指向 RAM 程序区的程序入口地址，Boot 完毕后，DSP 进入正常工作。DSP 程序正常运行后应采取手段保护 FLASH 内部的程序，以备下次再开机或重新复位时 Boot 用。如果 FLASH 空间足够大，多余的空间可以作为外部数据区。

Boot 可以直译为 DSP 的脱离仿真器启动，或称为自举启动。而相应的在仿真环境下的启动是靠仿真器完成的，可以称为仿真启动。

思考题

(1) 什么是 Boot Loader?
(2) 进行在线写 FLASH 的硬件基础是什么?
(3) 什么是现场 Boot Loader 程序设计? 它的好处是什么?
(4) 如何编写体现时序机制的 C/C++语句?

6.2 在线 Boot Loader

6.2.1 Boot Loader 概念

一个可编程的数字化芯片如 VC5402 等从上电复位到进入工作状态前一瞬间的这个阶段称为 Boot 阶段。有些简单的可编程数字化芯片，当上电复位后，它的程序指针自动指到一个固定的入口地址，程序设计者必须将程序可执行代码的首地址放在这个入口地址处。对于 DSP 芯片来说，它的内部 ROM 空间是有限的，生产商在这个小容量的 ROM 中固化了一些程序和查找表，对于 C5000 系列 DSP 来说，当上电复位后，程序指针自动指向 ROM 中的一个称为 Boot Loader 的小程序，这个程序会根据环境选用相应的 Boot Loader 方法，将外部 FLASH 中的程序搬移到 DSP 内部的 RAM 程序区中，并将程序指针移到执行程序的第一行处。

DSP 的 Boot Loader 程序是由生产商固化和升级管理的，对用户来说，ROM 区一般是不能改变的。用户可以使用 ROM 中的正弦查找表以及 A 律和 μ 律压扩表，但最好不要使用 Boot Loader 内部的函数。因为生产商可能随时会完善 Boot Loader 程序以及其中的函数。相同系列的 DSP 芯片，这个程序也可能有所不同。对于 VC5402 来说，ROM 的编址是从 F000h 至 FFFFh，具体的内容如表 6-1 所示。

表 6-1　VC5402 片上 ROM 的内容

地址范围	F000-F7FF	F800-FBFF	FC00-FDFF	FE00-FEFF	FF00-FF7F	FF80-FFFF
内容	保留	Boot Loader	μ 律	A 律	正弦查找表	向量表

可见 Boot Loader 程序很短小，主要完成一些数据的搬移和程序的重定位等工作。如果用户打算自己设计 Boot Loader 程序，必须与生产商联系重新烧写 ROM。由生产商提供的 Boot Loader 程序中提供了多种 Boot 的方法，用户可以根据自己的需要选用。

6.2.2 Boot Loader 模式

按照 Boot 时程序由外部 FLASH 等存储器进入到 DSP 片上 RAM 的通道不同分为很多种 Boot Loader 的模式。一般地，C5000 系列均支持 HPI、并口、串行口等模式，有的还支持通用 I/O 口等模式；按照数据进入 DSP 时的字长又分为 8 位和 16 位模式。DSP 上电后，会根据片上的环境采取相应的 Boot Loader 模式进行 Boot。

对于 VC5402，Boot 的过程如图 6-1 所示（图 6-1 直接引自 TI 公司的芯片资料）。从图 6-1 中可以看出，上电复位后，VC5402 首先判断 INT2 标志位（INT2 是外部中断 2，位于 IMR 和 IFR 寄存器中，IMR 是响应和屏蔽中断用的，IFR 是中断标志用的）。当 INT2 被激活时，则采取 HPI 的 Boot 模式；否则不使用 HPI 模式，进一步判断 INT3 标志位，根据 INT3 标志位的情况决定是否采用串口 EEPROM 的 Boot 模式。若这个条件不满足时，接着去读取 I/O 空间的 FFFFh 地址，当该地址内容为一有效的地址时，采用并口 Boot 模式；当该地址无效时，接着去读数据空间的 FFFFh 地址处，当该地址内容为一有效的地址时，仍采用并口 Boot 模式。若不满足条件时，进一步根据需要去采用其它的 Boot 模式，直到全部 Boot 模式遍历完毕。若仍没有满足要求的，则 Boot Loader 程序又会从 HPI 模式开始进行第二次的遍历，直到找到一种能够执行的 Boot 模式为止。

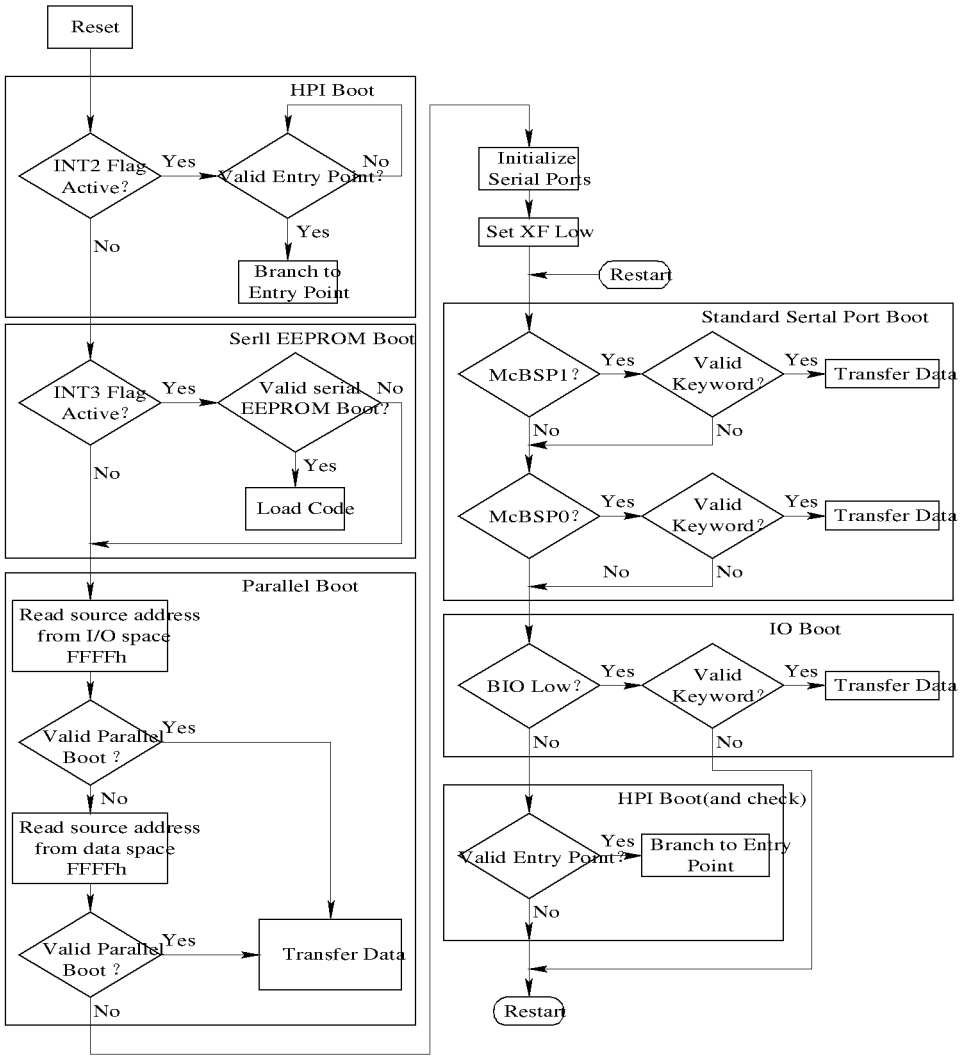


图 6-1　VC5402 的 Boot Loader 流程图

注：为方便读者阅读，作者给出图 6-1 中的部分译文，供参考。

(1) Reset　硬件上电复位
(2) INT2 Flag Active　INT2 标志激活
(3) INT3 Flag Active　INT3 标志激活
(4) Read source address from I/O space FFFFh　从 I/O 空间 FFFFh 地址处读数据
(5) Valid Parallel Boot　有效的并口 Boot 模式
(6) Read source address from data space FFFFh　从数据空间 FFFFh 地址处读数据
(7) Valid Parallel Boot　有效的并口 Boot 模式
(8) Valid Entry Point　有效的入口地址
(9) Branch to Entry Point　跳转到这个入口地址
(10) Valid serial EEPROM Boot　有效的串行 EEPROM Boot 模式
(11) Load Code　装入程序
(12) Transfer Data　传输数据
(13) Initialize Serial Ports　初始化串行口
(14) Set XF Low　设置 XF 为低
(15) McBSP1
(16) McBSP0
(17) BIO Low　BIO 为低
(18) Valid Entry Point　有效的入口地址
(19) Restart　重新开始
(20) Restart　重新开始
(21) Valid Keyword　有效关键字
(22) Valid Keyword　有效关键字
(23) Valid Keyword　有效关键字
(24) Branch to Entry Point　跳转到这个入口地址
(25) Transfer Data　传输数据
(26) Transfer Data　传输数据
(27) Transfer Data　传输数据
(28) Serial EEPROM Boot　串行 EEPROM Boot
(29) Parallel Boot　并行 Boot
(30) Standard Serial Port Boot　标准串行口 Boot
(31) HPI Boot (2nd check)　HPI Boot(第二次重复检查)

从图 6-1 可以看出，VC5402 有 5 种 Boot Loader 模式。访问是具有优先级区别的，按从高到低的顺序依次为：HPI 模式（增强主机接口）、串口 EEPROM 模式、并口模式、标准串行口模式和通用 I/O 口模式。对于 SY-5402EVM 板来说，硬件连接要求最好选用并口模式进行 Boot。并口模式也是较常用的一种模式。

6.2.3　并口 Boot Loader 方法

本节针对 VC5402 具体介绍并口 Boot Loader 的方法，对于 C5000 系列的其他芯片也是

同样的原理。

VC5402 上电复位后，当 INT2 和 INT3 没有中断触发时（INT2 和 INT3 是外部管脚，让这两个管脚悬空即可），MP/MC=0，Boot Loader 程序去读取 I/O 空间的 FFFFh 地址。当这个地址包含了有效的地址内容时，Boot 程序将该地址中的内容当作是 FLASH 中程序的首地址，Boot 程序会从这个首地址开始读取数据并复制到内部程序空间中去。FLASH 中的程序是有规律排列的，常被称为 Boot 表，Boot 表就是所有要下载到 DSP 内部程序区的程序代码。当 I/O 空间的 FFFFh 不含有有效的内容时，Boot 程序会读取数据空间的 FFFFh 地址处的内容。如果该内容合法，Boot 程序将该内容作为 FLASH 中 Boot 表的首地址，去进行 Boot Loader 过程。当数据区 FFFFh 处的内容非法时，VC5402 将不采用并口 Boot 模式，进入串行口 Boot 方式的判断。

图 6-2 为 Boot 程序运行时访问 Boot 表及数据传输的情况（该图原文引自 TI 公司的技

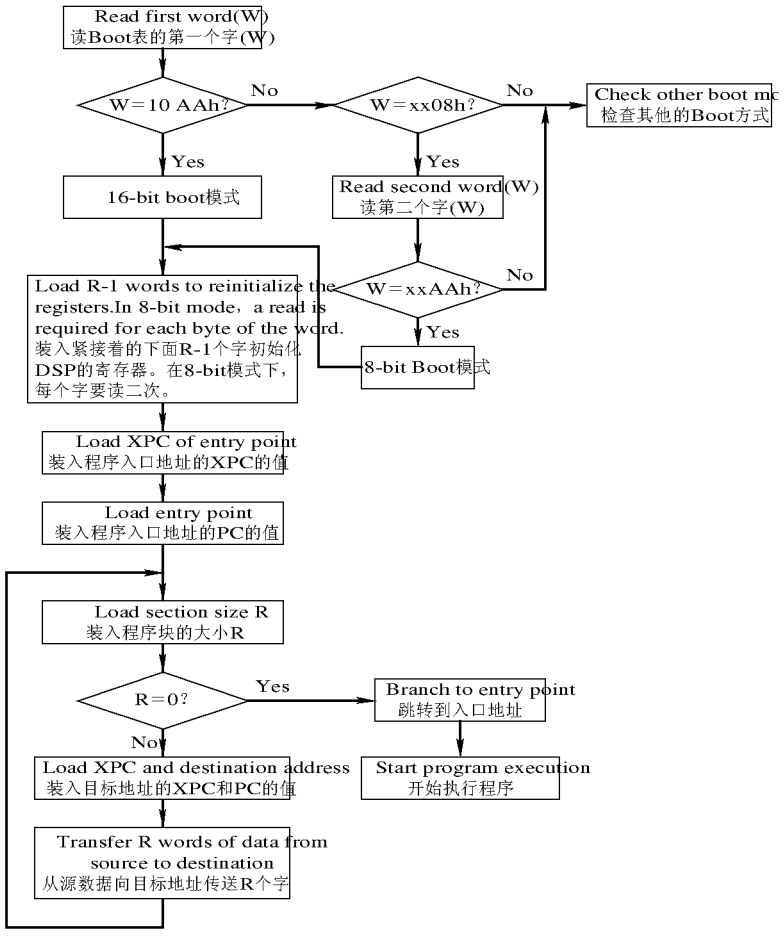


图 6-2　VC5402 的 Boot 程序运行流程图

术资料）。除了 HPI 模式不需要 Boot 表之外，其它模式均需要建立 Boot 表。如图 6-2 所示，读取 Boot 表的过程是：首先读取 Boot 表的第一个字，当该字为 0x10AA 时，采用 16 位的读取模式；当第一个字的后面 8 位是 0x08，下一个字的后面 8 位是 0xAA 时，采用 8 位的读取模式；接着读取下面的 R-1 个字初始化寄存器的值（根据不同的模式 R 具有不同的值，对于并口，R=3）；下面为程序完成 Boot 之后的入口偏移指针 XPC 和指针 PC；再下面的结构是相似的，依次为区段的长度、搬移到目的程序区的 XPC 和 PC 以及所需搬移的区段内容；Boot 程序将依次将各个区段搬移到内部 RAM 程序区中；最后，全部搬移完成后，程序指针会跳转到程序入口处，DSP 进入工作状态。16 位模式下通用 Boot 表的结构如表 6-2 所示。

表 6-2　16 位模式下通用 Boot 表的结构

序　号	内　容　及　意　义
1	10AA（16 位的存储格式）
2	这一个字的值是用来设置内部寄存器的值
…	…
R	这一字及以上的字的值用于设置内部寄存器
R+1	完成 Boot 之后的程序入口地址的偏移地址 XPC
R+2	完成 Boot 之后的程序入口地址 PC
R+3	第一个程序段的大小（N1）
R+4	第一个程序段要装入的内部 RAM 区的偏移地址
R+5	第一个程序段要装入的内部 RAM 区的地址
R+6	第一个程序段的第一个字（16 位）
…	…
R+N1+5	第一个程序段的最后一个字
R+N1+6	第二个程序段的长度（N2）
R+N1+7	第二个程序段要装入的内部 RAM 区的偏移地址
R+N1+8	第二个程序段要装入的内部 RAM 区的地址
R+N1+9	第二个程序段的第一个字
…	…
R+N1+N2+8	第二个程序段的最后一个字
…	…
…	…
…	…
R+N1+N2+8+LN（LN 为从第三个程序段开始到最后一个程序段的最后一个字为止的长度）	最后一个程序段的最后一个字
R+N1+N2+8+LN+1	0x0000（Boot 表以 0x0000 结尾）

下面给出了本章程序实例用到的并口模式 Boot 表的全部内容：

10 AA　——代表 16 位模式
7F FF 88 02　——这两个字分别用来设置 SWWSR 和 BSCR
00 00　——程序入口偏移地址
01 00　——程序入口地址
01 30　——第一个程序段的长度
00 00　——第一个程序段的偏移地址
01 00　——第一个程序段的地址
77 18 31 00 6B F8 00 18　——下面为第一个程序段的长为 0x130 的内容
03 FF 68 F8 00 18 FF FE F7 B8 F7 BE F6 B9 F4 A0 F6 B7 F6 B5 F6 B6 F0 20
02 30 F1 00 00 01 F8 4D 01 2B F6 B8 F0 20 02 30 F0 73 01 25 7E F8 00 12
F0 00 00 01 47 F8 00 11 7E 92 00 F8 00 11 F0 00 00 01 7E F8 00 11 F0 00
00 01 6C 89 01 1A F7 B8 EE FC F0 20 FF FF F1 00 00 01 F8 4D 01 3F F2 73
01 39 4E 02 F4 95 F5 E3 56 02 7E 00 11 00 FA 4C 01 37 6B 03 00 01 F6 B8
EE 04 F0 74 01 FB F0 74 01 45 4A 11 4A 16 72 11 35 20 10 F8 00 11 FA 45
01 5B F4 95 EE FF 48 11 F0 00 35 00 88 16 F4 95 F4 95 10 EE FF FF F4 E3
6C E9 FF FF 01 55 10 F8 35 21 F8 45 01 62 10 F8 35 21 F4 E3 F0 74 01 7F
EE 01 8A 16 8A 11 FC 00 F7 B8 E9 20 4A 11 09 F8 35 20 F8 4E 01 73 F2 73
01 7D F4 95 E8 01 72 11 35 20 49 11 80 E1 35 00 F3 00 00 01 E8 00 81 F8
35 20 8A 11 FC 00 F4 95 F0 73 01 80 4A 11 77 11 00 58 77 12 00 58 76 81
00 00 96 0F F8 20 01 8F 96 0F F8 30 01 8C 76 81 9F F7 77 11 00 1D 76 81
00 A0 77 11 00 28 76 81 7F FF 77 11 00 2B 76 81 00 01 77 11 00 29 76 81
88 02 77 11 00 48 76 81 00 00 77 11 00 49 76 81 00 21 77 11 00 48 76 81
00 01 77 11 00 49 76 81 02 01 77 11 00 48 76 81 00 02 77 11 00 49 76 81
00 40 77 11 00 48 76 81 00 03 77 11 00 49 76 81 00 00 77 11 00 48 76 81
00 04 77 11 00 49 76 81 00 40 77 11 00 48 76 81 00 05 77 11 00 49 76 81
00 00 77 11 00 48 76 81 00 06 77 11 00 49 76 81 00 00 77 11 00 48 76 81
00 07 77 11 00 49 76 81 00 00 77 11 00 48 76 81 00 08 77 11 00 49 76 81
00 00 77 11 00 48 76 81 00 09 77 11 00 49 76 81 00 00 77 11 00 48 76 81
00 0E 77 11 00 49 76 81 00 0C 8A 11 FC 00 4A 11 F0 74 01 82 77 1A 04 00
F0 72 02 29 77 11 00 48 77 12 00 49 76 81 00 00 96 0E F8 30 02 0E 96 0E
F8 20 02 0B 77 11 00 41 F7 B8 71 81 35 22 F0 20 FF FE 68 F8 35 22 FF FE
18 F8 35 22 77 11 00 48 80 F8 35 23 76 81 00 01 96 0E F8 30 02 26 96 0E
F8 20 02 23 77 11 00 43 70 81 35 23 77 11 00 07 F2 73 01 FE 6A 81 20 00
00 07　——第二个程序段的长度
00 00　——第二个程序段的偏移地址
02 30　——第二个程序段的地址
00 01 35 20 00 00 00 01 35 21 00 00 00 00　——第二个程序段的长为 0x07 的内容
00 64　——第三个程序段的长度

```
00 00                                          ——第三个程序段的偏移地址
00 80                                          ——第三个程序段的地址
F2 73 01 00 77 18 00 80                        ——以下为第三个程序段长为 0x64 的内容
F4 EB F4 95 F4 95 F4 95 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F4 EB F4 95 F4 95 F4 95 F4 EB
F4 95 F4 95 F4 95 F4 EB F4 95 F4 95 F4 95 F4 EB F4 95 F4 95 F4 95 F4 EB
F4 95 F4 95 F4 95 F4 EB F4 95 F4 95 F4 95 F4 EB F4 95 F4 95 F4 95 F4 EB
F4 95 F4 95 F4 95 F4 EB F4 95 F4 95 F4 95      ——第三个程序段结束，本实例中仅有三个程序段
00 00                                          ——Boot 表以 0x0000 为结束字
```

下面介绍如何生成这个 Boot 表，将第三章中的实例生成的可执行文件 user_audio.out 作为输入文件，利用 CCStudio 自带的 Hex500 程序包在命令行模式下生成这个文件，文件名可以自己设置，在此定名为 user_audio.hex。

实际的过程是首先制作一个 CMD 文件，文件名为 out2hex.cmd，内容如下：

```
user_audio.out    /* hex500 out2hex.cmd*/
-a
-map    user_audio.mxp
-o user_audio.hex
-bootorg PARALLEL
-e 0x100    /* -e    _c_int00*/
-boot
-swwsr 0x7fff
-bscr 0x8802
-memwidth 16/* default 16*/
-romwidth 16
```

程序解释如下：第一行为输入的 .out 文件名；第二行为输出 ASCII 的十六进制格式；第三行为产生名为 user_audio.mxp 的 MAP 表，可以从 MAP 表中找到程序的入口地址；第四行指定输出文件名；第五行为采用并口 Boot 模式；第六行定义程序入口地址，这个入口地址不能确定时可以参考 MAP 表；第七行为生成一个 Boot 表；第八行和第九行为指定 SWWSR 和 BSCR 寄存器的值；剩下两行为指定存储器字长。

将 hex500.exe、out2hex.cmd 和 user_audio.out 放在同一个目录下，执行命令行：

```
hex500    out2hex.cmd
```

即可以得到 user_audio.hex 文件，同时还得到一个 user_audio.mxp 文件。user_audio.mxp 文件的内容如下，从这个 MAP 表中可以很容易地找到程序入口地址。

以下为 user_audio.mxp 文件的内容：

```
*******************************************************************************
TMS320C54x COFF/Hex Converter                                        Version 3.70
*******************************************************************************

INPUT FILE NAME: <user_audio.out>
OUTPUT FORMAT:      ASCII-Hex

PHYSICAL MEMORY PARAMETERS
    Default data width:         16
    Default memory width :  16
    Default output width :    16

BOOT LOADER PARAMETERS
    Table Address: 0x0000, PAGE 0
    Entry Point     : 0x0100

REGISTERS
    spc:                00000018
    spce:               00000003
    arr:                00000800
    bkr:                00000010
    tcsr:               00000000
    trta:               00000001
    swwsr:              00007fff
    bscr                00008806


OUTPUT TRANSLATION MAP
--------------------------------------------------------------------------------
00000000..00003fff    Page=0    Memory Width=16    ROM Width=16    "FLASH"
--------------------------------------------------------------------------------
    OUTPUT FILES: user_audio.hex [b0..b15]

    CONTENTS: 00000000..000001a9      BOOT TABLE
                                  .text : dest=00000100    size=00000130    width=00000002
                                 .cinit : dest=00000230    size=00000007    width=00000002
```

上面文件中，加阴影的为程序的入口地址。如果生成 Boot 表时，设定的地址与这个地址不相同，则必须修改-e 选项，重新生成一个新的 Boot 表。关于使用 Hex500 将.out 文件转

换为 hex 格式的文件，从而生成 Boot 表的方法，可以参考 CCStudio 提供的在线“帮助”文档，这里不再介绍。

关于生成 Boot 表的补充事项，就是关于输入文件 user_audio.out 的问题。因为 .out 文件本身也有两个版本，一般地编译方法生成的是旧的版本，这个旧的版本在使用 hex500 时可能有时会出错，所以在编译用于 Boot 表的.out 文件时应使用编译选项 –v548。设置方法为：进入 CCStudio/Project/Build Options，如图 6-3 所示；并进入到连接选项卡中设置初始化模式为 Run-time Autoinitialization，如图 6-4 的加亮条。在图 6-3 中可以设置优化。

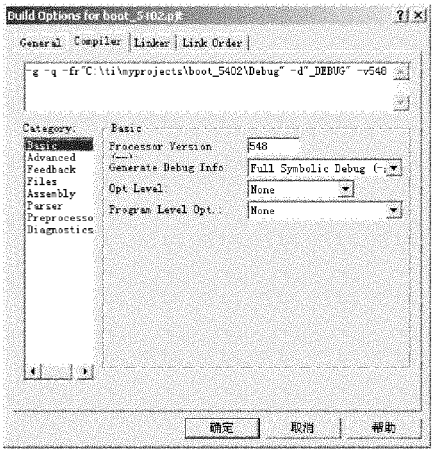


图 6-3　编译选项卡

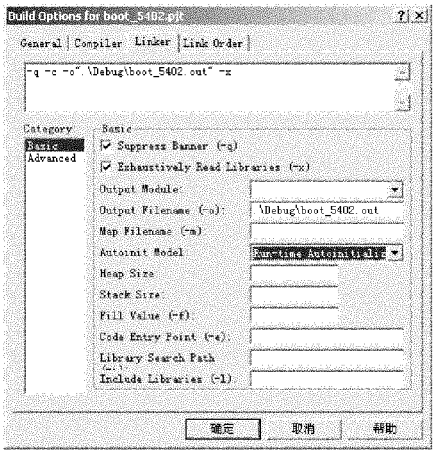


图 6-4　连接选项卡

6.2.4　现场 FLASH 编程

现场 FLASH 编程是指通过仿真器和 JTAG 口借助 DSP 芯片将 Boot 表写入 FLASH 的过程。这个过程不需要将 FLASH 从 DSP 功能板上取下来，更不需要借助于编程器等其它的烧写程序的设备，而只限于仿真环境下即可。

现场 FLASH 编程又称为在线 FLASH 编程，这对整个系统有一定的要求，即必须将 FLASH 与 DSP 芯片的 HPI 口或并口或串口等数据通道管脚相连接。在线 FLASH 编程使得系统的灵活性大大加强，下载软件到 DSP 功能板上很方便，还可以通过加载不同的软件完成不同的功能，也可以根据需要不断地升级软件，从而增强了硬件平台的功能，提高了利用率。

SY-5402EVM 板上选用了 SST39VF400FLASH 芯片，是一款常用的高档的 FLASH，具有代表性。第 6.3 节我们将以向 SST39VF400 写入程序为例，介绍现场 FLASH 编程的方法。

6.3　Boot 硬件基础

6.3.1　SY-5402EVM 板存储器的设置

SY-5402EVM 板上有两片 IS61LV6416 和一片 SST39VF400。两片 IS61LV6416 为

64 KW(W=Word)的 RAM，分别用于扩展程序区和数据区，而 SST39VF400 可用于扩展程序区或是数据区，或是用于 I/O 空间，根据具体设置而言。

将 SY-5402EVM 上的 JP12、JP13、JP14 和 JP15 等跳线跳到低电位。现在主要关心的是 FLASH 的映射，从三意电子提供的 CPLD 逻辑中可以总结出下面的映射关系：

对于 I/O 空间，FLASH 映射到 0x8000 至 0xFFFF；

对于数据空间，FLASH 映射到 0x0000 至 0xFFFF，但是用户仅能使用 0x4000 至 0xFFFF（DROM=0 时）；

对于程序空间，FLASH 映射到 0x20000 至 0x5FFFF。

因此，对于 I/O 空间和数据空间而言，地址范围为 0x8000 至 0xFFFF 的区域是重合的，这部分空间可以通过 I/O 口访问，也可以通过数据区访问。于是，当把 Boot 表存放在此区间内时，可以有两种在线 FLASH 编程的方法，一种是通过专用的 I/O 口指令来完成；另一种是通过对数据空间的访问来完成。本章示例程序的 Boot 表就是存放在 FLASH 的 0x8000 到 0x81A9 空间内。如果用户想将 Boot 表写入到 FLASH 的 0x4000 至 0x7FFF 空间内，则只能通过访问数据空间来完成。

6.3.2　SST39VF400 介绍

各种 FLASH 芯片的编程原理大同小异，只要掌握一种 FLASH 的编程方法即可。

下面以 SST39VF400 为例介绍 FLASH 芯片的擦除、编程（写入）、读取的方法及注意事项。一些图片直接引自芯片资料。

为了简便，以下将 SST39VF400 简称为 VF400。它是一个低功耗 FLASH，工作在 2.7 V 至 3.6 V 电压下，存储容量为 256 KW（1W=16 bit），其中的数据可以保存 100 年以上，可重复编程次数高达 10 万次。VF400 的功能图如图 6-5 所示。其中，A17 至 A0 为外部地址管脚，DQ15 至 DQ0 为 16 条数据线，CE#为片选控制管脚（低有效），OE#为输出控制管脚（低有效），WE#为写入控制管脚（低有效）。

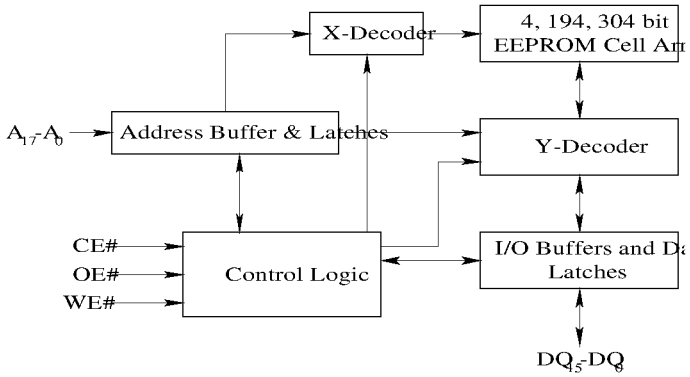


图 6-5　VF400 的功能框图

1) 擦除

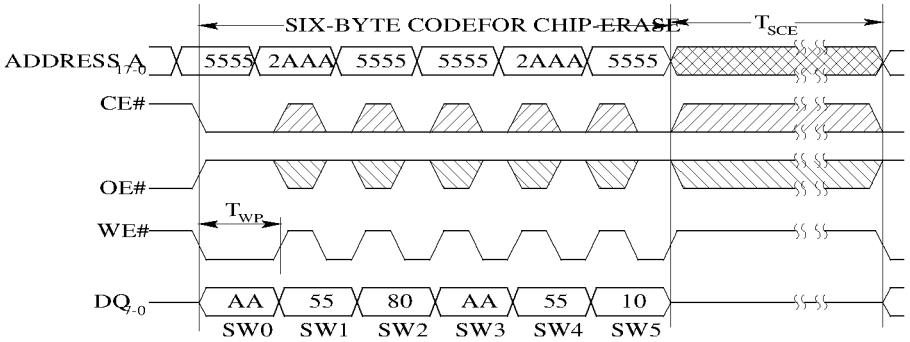
VF400 提供了三种擦除方式：其一是按扇区擦除，每扇 2 KW，共 128 扇；其二是按块擦除，每块 32 KW，共 8 块；其三是整片快速擦除。前两种方法为写入小程序时采用的部分擦除方法，后一种方法适用于大程序编程写入 FLASH。本章的示例中给出了第三种擦除方法。

擦除步骤如表 6-3 所示。

表 6-3　整片擦除步骤

步骤	1	2	3	4	5	6
地址(0x)	5555	2AAA	5555	5555	2AAA	5555
本示例地址(0x)	D555	AAAA	D555	D555	AAAA	D555
数据(0x)	AA	55	80	AA	55	10

在整片擦除过程中，依次向表 6-3 所示的地址处写入（或称编程）表中所示的数据。在 SY-5402EVM 板上，访问上表中的地址实际上是不可能的，因为 I/O 空间中的 FLASH 映射部分在 0x8000 至 0xFFFF。好在 VF400 在擦片时只能认到 A0～A14，A15、A16 和 A17 都被忽略了，所以在实际编程时采用的地址是表 6-3 所示的“本示例地址”一栏中的地址。擦片时的时序如图 6-6 所示。



注：本芯片也支持由 CE 管脚控制的片上擦除操作。WE 信号和 CE 信号在满足时序情况下交替有效，本图中仅考虑 WE 信号的作用

图 6-6　VF400 整片擦除的时序（由 WE#控制）

在完成表 6-3 所示的六步以后，在第六步的 WE 的上升沿（边沿触发）VF400 进入内部工作状态，这时不需要用户干预，这个过程大约需要 70 ms。用户可以通过判断 DQ6 和 DQ7 的状态来了解是否已经完成擦除：当擦除正在进行时，DQ6 脚上出现 0 和 1 两种电平的跳跃，当擦除完成后，DQ6 稳定到一个电位上；当擦除在进行时，DQ7 为 0 电位，当擦除完成后，DQ7 为 1 电位，通常采用 DQ7 来判断有没有擦除完毕。实际在对 FLASH 编程（或称写入）时 DQ6 和 DQ7 也是判断标志：当编程正在进行时，DQ6 跳变，常用 DQ6 的稳定来判断数据有没有写入到 FLASH 中。本实例中采用了这种方法。当编程正在进行时，DQ7

上出现与真实数据同一位上电位相反的电位值；当擦除完毕后，DQ7 上为真实的电位。其实，用 DQ7 来判断编程和用来判断擦除与用 DQ6 一样是很方便的。DQ6 称为 Toggle Bit，DQ7 称为 Data#Polling。图 6-7 为使用 DQ6 和 DQ7 进行判断的流程图。如图 6-7 所示，擦除的六步（或者编程的四步）完成之后，依次读出两个字（地址任意），判断这个字的第六位是否相同，若相同就表明已经擦除（或编程）完毕，退出；如果不相同，则返回去再依次读两个字，直到判断所读两个字的第六位相同为止。

对于 DQ7 来说，当判别擦除情况时，读一个字判断它的 DQ7 是否为 1，不为 1 时返回去再读一个字（地址任意），直到读出的字的 DQ7 为 1 跳出循环。判别编程情况时，读一个字，并将这个字的 DQ7 位与写入数据的 DQ7 位进行比较：如果相同，则跳出循环，表明数据已经写入 FLASH；不相同时，继续循环。

2) 编程

编程只有一种方法，但是有两个时序控制：一种是由 WE#来控制的，另一种是由 CE#来控制的。由 WE# 控制的方法常用，对于 SY-5402EVM 板来说，只能用 WE# 来控制。

编程步骤有四步，如表 6-4 所示。

表 6-4　编 程 步 骤 表

步骤	地址	本示例地址	数据
1	0x5555	0xD555	0xAA
2	0x2AAA	0xAAAA	0x55
3	0x5555	0xD5555	0xA0
4	编程地址	编程地址	编程数据

在表 6-4 中同样是采用“本示例地址”向 FLASH 中写入数据。完成上表的四步后，也要根据图 6-7 判断 DQ6 或 DQ7 的情况来了解编程过程是否完成。在本章示例程序中，给出了用 DQ6 来判断编程过程是否结束的 C 语言编程方法及源代码。编程的时序如图 6-8 所示，DQ6 和 DQ7 的时序如图 6-9 和图 6-10 所示。

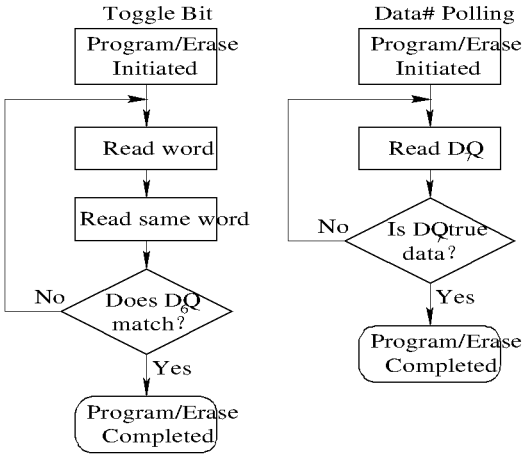


图 6-7　DQ6 和 DQ7 的判断流程图

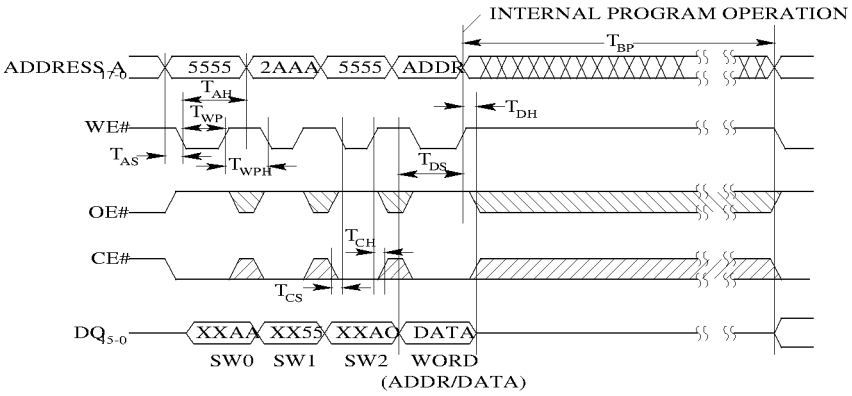


图 6-8　VF400 编程时序图（由 WE 控制）

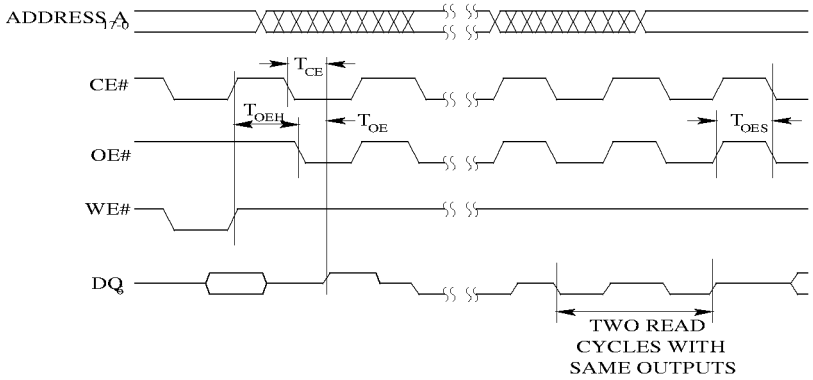


图 6-9　DQ6 的时序图

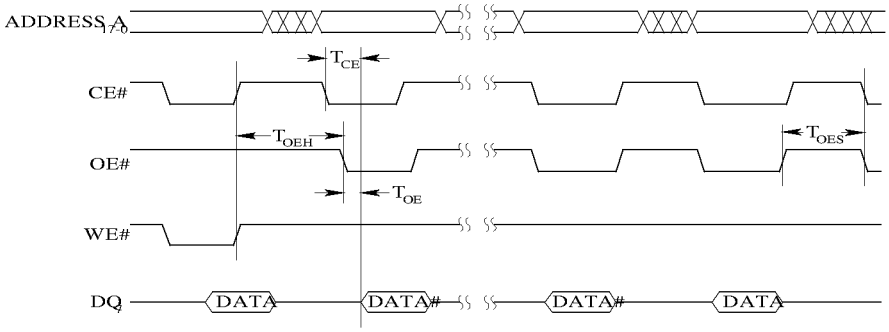


图 6-10　DQ7 的时序

3) 读取

VF400 读取数据是很方便的，按图 6-11 的时序要求，在不少于 5 ns 的时序下，数据总会安全地读出。

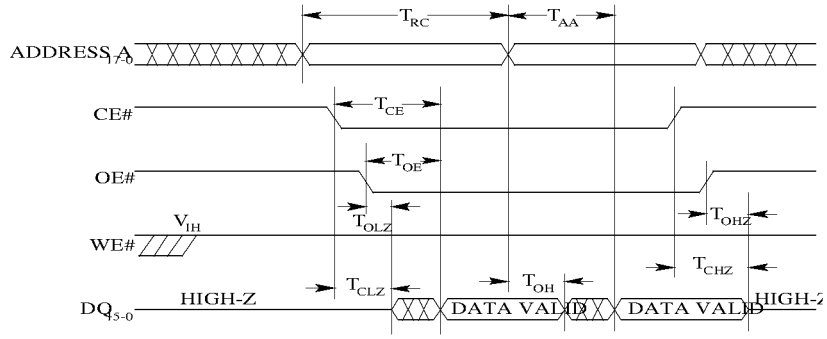


图 6-11　VF400 读时序图

进入正常工作状态的方法：VF400 可以进行读写的状态为正常工作状态。此外，VF400 还提供了几种保护片上数据的软方法和硬方法以及其它的工作状态。进入正常工作状态有两种方法，现仅介绍一种方法，其步骤如表 6-5 所示。

表 6-5　VF400 进入正常工作状态的步骤

步　　骤	1	2	3
地址（0x）	5555	2AAA	5555
本示例程序地址（0x）	D555	AAAA	D555
数据（0x）	AA	55	F0

对于地址访问，采用 I/O 操作时需采用本示例程序地址(如果读者使用数据区向 FLASH 写入数据，0x5555 是可以的, 但是 0x2AAA 必须改为 0xAAAA 才可以)，时序如图 6-12 所示。

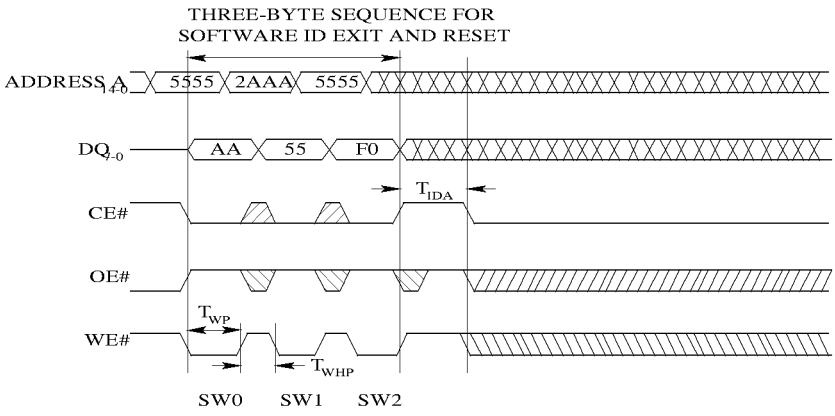


图 6-12　VF400 进入正常工作状态的时序

这一步不是必需做的，在本章的示例程序中给出了进入正常工作状态的方法和源程序。

综上所述，可以总结出写 VF400 这块 FLASH 芯片的步骤如下：

? 第一步：编程使 FLASH 进入正常工作状态（如果 FLASH 本来就处于正常工作状态，这一步可以省略）。

? 第二步：将要编程的 FLASH 空间擦除。有三种擦除方法，本章程序采用了整片擦除。一般地，写数据到 FLASH 内部时，通常应该擦除这部分的内容；否则，可能会出现误码。因此，这一步是不可缺少的。

? 第三步：将 Boot 表写入到 FLASH 中。写入过程是由 CCStudio 的软件控制的，即写入的详细地址由软件控制。

? 第四步：将写入 FLASH 中的内容读出，进行读出校验。对 SY-5402EVM 板上的 VF400 来说，这一步可以省略，因为通过 View - Memory 菜单可以查看到 I/O 空间或数据空间的相应地址处的数据。

6.4 程 序 设 计

6.4.1 编程准备工作

为了让读者更好地理解编程的详细过程，这里列出一些必要的准备工作。本节介绍的示例程序是在 SY-5402EVM 板上调试通过的，可以直接应用于实际 DSP 中。

? 准备一：使用 hex500 可执行文件、out2hex.cmd 参数文件和 user_audio.out 文件生成一个 Boot 表，命名为 user_audio.hex。这一准备工作前面有详细的介绍，这里要补充说明的是 user_audio.out 文件最好为 Release 版本，而不是 Debug 编译版本。因为 Release 版本是发行版本，将其中的调试信息去掉了，所以程序要比 Debug 版本小一些。

? 准备二：编制一个 C 程序将 user_audio.hex 文件中的数据取出，并形成一个数组文件的形式，即将前面列出的 user_audio.hex 文件转换为 boot_dat.h 中的形式。这一步转换是为了更好地介绍和理解 Boot Loader 程序设计方法，读者熟练后可以直接从 user_audio.hex 文件中读取内容。下面列出了转换程序和转换后的文件。

//下面为转换程序，已在 Borland C3.1 或 Visual C++6 下运行通过。

```
#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
#include "iostream.h"

void main()
{
    char *str;
    FILE *fp;
    FILE *fp2;
```

```
fp=fopen("d:\\user_audio.hex","r");
fp2=fopen("d:\\tem_au.dat","w");
int i=0;
int j=1;
fgets(str,2,fp);
cout<<str<<endl;

fgets(str,2,fp);
cout<<str<<endl;
while(!feof(fp))
{
    fgets(str,2,fp);
    if(j%6==1)
    {
        fputs("0x",fp2);
        fputs(str,fp2);
    }
    if(j%6==2)
    {
        fputs(str,fp2);
    }
    if(j%6==3)
    {
        //fputs(str,fp2);
    }
    if(j%6==4)
    {
        fputs(str,fp2);
    }
    if(j%6==5)
    {
        fputs(str,fp2);
    }
    if(j%6==0)
    {
        fputs(",",fp2);
        if(i>10)
        {
            fputs("\n",fp2);
```

```
                    i=0;
                }
                i++;
            }
            j++;
        }
        fclose(fp);
        fclose(fp2);
        getch();
}
```

上面的程序用 switch 语句改写也许会更好一些，这时生成的 tmp_au.dat 文件即为数组中的元素。添加数组定义成为下面的文件：

```
//boot_dat.h  下载到 FLASH 中的全部数据

u16 flash_data[426]=
{0x10AA,0x7FFF,0x8802,0x0000,0x0100,0x0130,0x0000,0x0100,0x7718,0x3100,0x6BF8,
0x0018,0x03FF,0x68F8,0x0018,0xFFFE,0xF7B8,0xF7BE,0xF6B9,0xF4A0,0xF6B7,0xF6B5,
0xF6B6,0xF020,0x0230,0xF100,0x0001,0xF84D,0x012B,0xF6B8,0xF020,0x0230,0xF073,
0x0125,0x7EF8,0x0012,0xF000,0x0001,0x47F8,0x0011,0x7E92,0x00F8,0x0011,0xF000,
0x0001,0x7EF8,0x0011,0xF000,0x0001,0x6C89,0x011A,0xF7B8,0xEEFC,0xF020,0xFFFF,
0xF100,0x0001,0xF84D,0x013F,0xF273,0x0139,0x4E02,0xF495,0xF5E3,0x5602,0x7E00,
0x1100,0xFA4C,0x0137,0x6B03,0x0001,0xF6B8,0xEE04,0xF074,0x01FB,0xF074,0x0145,
0x4A11,0x4A16,0x7211,0x3520,0x10F8,0x0011,0xFA45,0x015B,0xF495,0xEEFF,0x4811,
0xF000,0x3500,0x8816,0xF495,0xF495,0x10EE,0xFFFF,0xF4E3,0x6CE9,0xFFFF,0x0155,
0x10F8,0x3521,0xF845,0x0162,0x10F8,0x3521,0xF4E3,0xF074,0x017F,0xEE01,0x8A16,
0x8A11,0xFC00,0xF7B8,0xE920,0x4A11,0x09F8,0x3520,0xF84E,0x0173,0xF273,0x017D,
0xF495,0xE801,0x7211,0x3520,0x4911,0x80E1,0x3500,0xF300,0x0001,0xE800,0x81F8,
0x3520,0x8A11,0xFC00,0xF495,0xF073,0x0180,0x4A11,0x7711,0x0058,0x7712,0x0058,
0x7681,0x0000,0x960F,0xF820,0x018F,0x960F,0xF830,0x018C,0x7681,0x9FF7,0x7711,
0x001D,0x7681,0x00A0,0x7711,0x0028,0x7681,0x7FFF,0x7711,0x002B,0x7681,0x0001,
0x7711,0x0029,0x7681,0x8802,0x7711,0x0048,0x7681,0x0000,0x7711,0x0049,0x7681,
0x0021,0x7711,0x0048,0x7681,0x0001,0x7711,0x0049,0x7681,0x0201,0x7711,0x0048,
0x7681,0x0002,0x7711,0x0049,0x7681,0x0040,0x7711,0x0048,0x7681,0x0003,0x7711,
0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,0x0004,0x7711,0x0049,0x7681,0x0040,
0x7711,0x0048,0x7681,0x0005,0x7711,0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,
0x0006,0x7711,0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,0x0007,0x7711,0x0049,
0x7681,0x0000,0x7711,0x0048,0x7681,0x0008,0x7711,0x0049,0x7681,0x0000,0x7711,
0x0048,0x7681,0x0009,0x7711,0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,0x000E,
```

```
0x7711,0x0049,0x7681,0x000C,0x8A11,0xFC00,0x4A11,0xF074,0x0182,0x771A,0x0400,
0xF072,0x0229,0x7711,0x0048,0x7712,0x0049,0x7681,0x0000,0x960E,0xF830,0x020E,
0x960E,0xF820,0x020B,0x7711,0x0041,0xF7B8,0x7181,0x3522,0xF020,0xFFFE,0x68F8,
0x3522,0xFFFE,0x18F8,0x3522,0x7711,0x0048,0x80F8,0x3523,0x7681,0x0001,0x960E,
0xF830,0x0226,0x960E,0xF820,0x0223,0x7711,0x0043,0x7081,0x3523,0x7711,0x0007,
0xF273,0x01FE,0x6A81,0x2000,0x0007,0x0000,0x0230,0x0001,0x3520,0x0000,0x0001,
0x3521,0x0000,0x0000,0x0064,0x0000,0x0080,0xF273,0x0100,0x7718,0x0080,0xF4EB,
0xF495,0xF495,0xF495,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
0x0000,0x0000,0x0000,0x0000,0xF4EB,0xF495,0xF495,0xF495,0xF4EB,0xF495,0xF495,
0xF495,0xF4EB,0xF495,0xF495,0xF495,0xF4EB,0xF495,0xF495,0xF495,0xF4EB,0xF495,
0xF495,0xF495,0xF4EB,0xF495,0xF495,0xF495,0xF4EB,0xF495,0xF495,0xF495,0xF4EB,
0xF495,0xF495,0xF495,0xF4EB,0xF495,0xF495,0xF495,0x0000};
//0x0--0x1a9(0--425) : 8000--81a9
```

这个文件将包括在主程序文件中。

？准备三：将 SY-5402 的 JP12、JP13、JP14 和 JP15 跳线跳到低电位上，将 CLKMD1、CLKMD2 和 CLKMD3 设为 110（这是本书采用的设置，最好是掉电后改变设置）。

6.4.2　程序流程

由上述可见，Boot Loader 程序设计主要是如何在线写 FLASH。在这里，借助这个示例程序介绍一下如何用 C/C++语言进行时序机制的编程设计。在对 FLASH 进行写入和读出操作时，都伴随着两条辅助性的读出和写入操作，这些辅助性的操作均以 0xFFFE 地址为辅助地址，它们都是为写入和读出操作提供时序的，并不是真的要去操作这个 0xFFFE 地址。例如，先读 FLASH，再写入 FLASH，可以在 WE# 中出现一个下降沿；写完 FLASH 再接着读一次，可以在 WE# 中出现一个上升沿；这样真正写入 FLASH 的时序机制就满足要求了，而它前后的两个读数据动作仅是为它提供时序，没有别的用处。图 6-13 列出了本示例程序的总体流程图。

按图 6-13 编写程序，并生成可执行 .out 文件。在仿真环境下运行程序后，关闭仿真器和 SY-5402EVM 板的电源，将仿真器与 SY-5402EVM 板脱离；然后单独给 SY-5402EVM 板上电，用计算机向 SY-5402EVM 板送入音频信号，SY-5402EVM 板会立即正常工作，D3-LED 灯闪烁，同时可以从 SY-5402EVM 板音频输出口听到清晰的音乐。这标志着 Boot 过程的成功以及 Boot Loader 程序设计的完成。

在下面的 6.4.3 节中给出了工程文件的源代码，并作了进一步的解释。

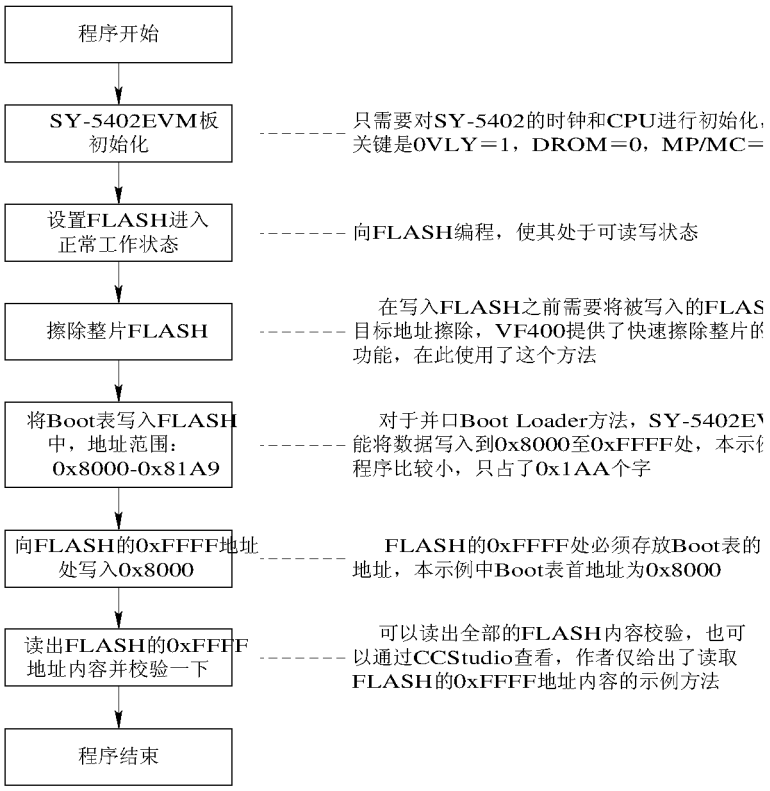


图 6-13　Boot Loader 程序流程图

6.4.3　程序源代码及分析

整个工程文件的组成如图 6-14 所示。从图 6-14 可以看出本工程文件中包括了主程序文件 boot_5402.c、配置文件 boot_5402.cmd、头文件 boot_5402.h、boot_dat.h、port_5402.h、port_func.h 和库文件 rts.lib，（需要注意的是，在本程序中没有使用中断向量表）。

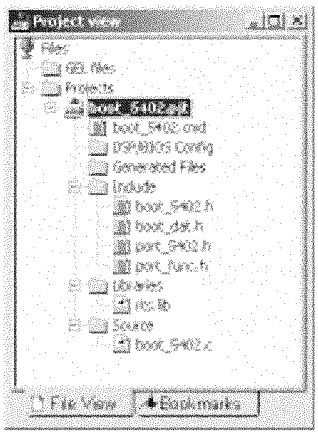


图 6-14　工程文件管理器

库文件是 CCStudio 提供的，前面已经介绍了 boot_dat.h 的全部内容，这里给出其他源程序的内容。

```
//下面为主程序文件 boot_5402.c
/*****************************/
/* filename: boot_5402.c          */
/* Author:ZhangYong               */
/* 2002-12                        */
/*****************************/
```

```
#include "port_5402.h"
#include "boot_5402.h"
#include "port_func.h"
#include "boot_dat.h"

/* Global Variables   */
u16   val;
// Main program
void main()
{
      u16 i;
      init_board();                    //init the VC5402  初始化 VC5402 CPU

      flash_ready();                   //使 FLASH 进入正常工作状态
      flash_erase();                   //整片擦除 FLASH
      //flash_write(0x8000,0x3ff6);
       for(i=0;i<=0x1a9;i++)           //写入 Boot 表
       {
              flash_write((0x8000+i),flash_data[i]);
       }
       flash_write(0xffff,0x8000);  //向 FLASH 的 0xFFFF 写入 0x8000
       val=flash_read(0xffff);         //读出 FLASH 的 0xFFFF 地址的值
       //while(1){}
}
```

//下面为头文件 boot_5402.h

```
//filename: boot_5402.h

typedef unsigned int u16;
typedef volatile ioport unsigned   IOPORT;

//--------- CPU ----------
#define     reg_ST1          0x0007

#define     PMST             0x001D
#define     SWWSR            0x0028
#define     SWCR             0x002B
#define     BSCR             0x0029
#define     CLKMD            0x0058
```

```
#define PMST_VAL        0x00A0          //interupt vectors from ox80
#define SWWSR_VAL       0x7fff
#define SWCR_VAL        0x0000//0x0001
#define BSCR_VAL        0x8802
#define CLKMD_VAL       0x9807

void init_board(void)       //初始化 VC5402 时钟和 CPU
{
    *(volatile u16 *)CLKMD = 0x0000;
    while(*(volatile u16 *)CLKMD & 0x0001){};
    *(volatile u16 *)CLKMD = CLKMD_VAL;

    *(volatile u16 *)PMST = PMST_VAL;
    *(volatile u16 *)SWWSR =SWWSR_VAL;
    *(volatile u16 *)SWCR =SWCR_VAL;
    *(volatile u16 *)BSCR =BSCR_VAL;
}
```

//下面为头文件 port_5402.h

```
ioport unsigned portaaaa;               //定义了 VC5402 的并口
ioport unsigned portd555;
ioport unsigned portfffe;
```

//下面为头文件 port_func.h

```
u16 u_tmp;
u16   u_dq7;
u16   u_dq61;
u16   u_dq62;
void flash_ready()                      //将 FLASH 调整为正常读写的工作状态
{
    u_tmp=*(u16 *)(0xfffe);
    *(u16 *)(0xd555)=0xaa;              //向 0xd555 写入 0xaa
    u_tmp=*(u16 *)(0xfffe);

    *(u16 *)(0xaaaa)=0x55;              //向 0xaaaa 写入 0x55
    u_tmp=*(u16 *)(0xfffe);
    *(u16 *)(0xd555)=0xf0;              //向 0xd555 写入 0xf0
    u_tmp=*(u16 *)(0xfffe);             //其它语句为提供时序用的
}
```

```
void flash_erase()                    //将 FLASH 整片擦除
{
    u_tmp=portfffe;                   //the following is erase entire flash --begin
    portd555=0xaa;                    //向 0xd555 写入 0xaa
    u_tmp=portfffe;

    portaaaa=0x55;                    //向 0xaaaa 写入 0x55
    u_tmp=portfffe;

    portd555=0x80;                    //向 0xd555 写入 0x80
    u_tmp=portfffe;

    portd555=0xaa;                    //向 0xd555 写入 0xaa
    u_tmp=portfffe;

    portaaaa=0x55;                    //向 0xaaaa 写入 0x55
    u_tmp=portfffe;

    portd555=0x10;                    //向 0xd555 写入 0x10
    u_tmp=portfffe;                   //其它语句为调整时序
    //u_tmp=portfffe;
    do
    {
        portfffe=0x11;
        u_dq7=portd555;
    }while((u_dq7 & 0x0080) == 0x0000);  //erase entir flash --end
                    //判断 DQ7，为 1 时，表示完成擦除，跳出循环；为 0 时，继续等待
}

void flash_write(u16 u_addr,u16 u_val)        //向 FLASH 中写入一个字
{
     u_tmp=*(u16 *)(0xfffe);                  //program flash --begin
    *(u16 *)(0xd555)=0xaa;                    //向 0xd555 写入 0xaa
    u_tmp=*(u16 *)(0xfffe);

    *(u16 *)(0xaaaa)=0x55;                    //向 0xaaaa 写入 0x55
    u_tmp=*(u16 *)(0xfffe);

    *(u16 *)(0xd555)=0xa0;                    //向 0xd555 写入 0xa0
    u_tmp=*(u16 *)(0xfffe);
```

```
    //port8000=0x123a;
    *(u16 *)(u_addr)=u_val;                    //写入编程数据
    u_tmp=*(u16 *)(0xfffe);

    do
    {
        *(u16 *)(0xfffe)=0x11;
        u_dq61=*(u16 *)(0x8000);
        *(u16 *)(0xfffe)=0x11;

        //u_dq62=port8000;
        u_dq62=*(u16 *)(0x8000);
        *(u16 *)(0xfffe)=0x11;
    }while((u_dq61 & 0x0040) != (u_dq62 & 0x0040));      //program flash --end OK
    //判断 DQ6 的值稳定与否，稳定了，退出；不稳定，循环等待
}

u16 flash_read(u16 u_addr)              //从 FLASH 中读出一个字
{                                       //这段程序里给出从 FLASH 中读出一个字的多种方法
    /* portfffe=0x11;                   //这里仅选用了一种，
    u_a=port8000;                       //其他的加了“/*  */”，也是可以用的
    portfffe=0x11;*/
    //OK

/* *(volatile unsigned int *)(0xFFFE)=0x11;
    u_b=*(volatile unsigned int *)(0x8000);
    *(volatile unsigned int *)(0xFFFE)=0x11;  */
    // OK
 /*  portfffe=0x11;
    val=portffff;
    portfffe=0x11; */
    u16 u_val;
    *(u16 *)(0xfffe)=0x11;
    u_val=*(u16 *)(u_addr);
    *(u16 *)(0xfffe)=0x11;

    return (u_val);

}
```

```
//下面为 boot_5402.cmd 配置文件
/*-m    boot_5402.map*/

MEMORY
{
      PAGE 0:       PROG(RWXI):      origin=0x100,      length=0x3000
                    VECS(RXI):       origin=0x80,       length=0x80
      PAGE 1:       DATA(RWI):       origin=0x3100,     length=0xF00
}

SECTIONS
{
      .text       {} >      PROG     PAGE 0
      .cinit      {} >      PROG     PAGE 0
      .pinit      {} >      PROG     PAGE 0
      .vectors    {} >      VECS     PAGE 0
      .stack      {} >      DATA     PAGE 1
      .bss        {} >      DATA     PAGE 1
      .const      {} >      DATA     PAGE 1
      .switch     {} >      DATA     PAGE 1
      .sysmem     {} >      DATA     PAGE 1
      .cio        {} >      DATA     PAGE 1
      .far        {} >      DATA     PAGE 1
      .data       {} >      DATA     PAGE 1
}

//上面的配置文件中多了一个.data
```

注：中文注释是后来加入的解释，实际编程时不用输入。

6.5 本章小结

本章给出的在线 Boot Loader 程序设计方法是相对固定的，虽然对于大的 Boot 表来说，可能要进行分块、分阶段 Boot Loader，但其基本编程方法是相同的。本程序虽然是针对 SY-5402EVM 板来介绍的，但对于 C5000 甚至 C6000 系列来说，原理是相同的。通过本章的学习，读者应基本掌握 Boot Loader 方法及其应用。

现在将本章开头提出的问题作一简要的回答。

Boot Loader 的概念在本章中多次解释，其本质是在 DSP 上电后完成将外部 FLASH 中的程序段读入 DSP 内部 RAM 程序区，并将程序指针指向程序运行时的首地址的过程。这样，完成 Boot Loader 之后，DSP 就可以独立地进行工作了。

在线写 FLASH 的硬件基础，就是 FLASH 本身编程的时序。简单地说，现场 Boot Loader 就是在仿真环境下向 FLASH 下载 Boot 表。

所谓的时序编程是相对于数据编程来说的。数据编程也就是数字信号处理算法的编程，是将输入的数据按某种算法或规则进行运算处理，再将得到的结果数据输出的过程。这个过程总体上是顺序完成的，需要一定的时间，但程序本身是不会去考虑时间的。数据编程可以独立地存在，如借助于通用计算机进行程序设计大都是数据编程。所以说，数据编程是面向 CPU 的一种方法。但时序编程往往不是独立地存在的，它是面向硬件接口的一种方法。时序编程为数据进入 CPU 进行数据编程做好各项准备工作。时序编程中的很多语句是为外部接口提供时序，不会产生数据流，数据流仅是其中的一部分语句。也就是说，时序编程中伴随有数据流的编程，并为数据流提供驱动时序。要想分清这种编程的概念，必须把握住时序编程仅是为数据输入/输出提供时序的安排，本身并不产生有效数据流，多是直接面向硬件接口的；数据编程本身是不考虑时序概念的（并行处理时也是一样），它重在数字信号处理算法的实现上。

习 题 六

1. 什么是 Boot Loader?
2. 什么是 Boot?
3. TMS320VC5402 的 ROM 中的内容是什么？
4. TMS320VC5402 的 ROM 是如何编址的？
5. DSP 的 Boot Loader 有哪些模式？
6. 简述并口 Boot Loader 的方法。
7. DSP 如何选用相应的 Boot Loader 方法？
8. 什么是 Boot 表？
9. 如何建立 Boot 表？
10. HPI 的 Boot 方式为什么不需要 Boot 表？
11. 什么是在线 Boot Loader?
12. 以 SST39VF400 为例简介 FLASH 的内部结构和访问方法。
13. 如何擦除 SST39VF400 中的数据？
14. 如何识别擦除或编程 SST39VF400 已经完毕？
15. 如何向 SST39VF400 中写入数据？
16. 写一段程序向 SST39VF400 的 0xFFFF 地址写入数据 0x8000。
17. 写一段程序从 SST39VF400 的 0xFFFF 地址读出数据。
18. 什么是时序编程？与数据编程有什么本质的区别？
19. 简述一下在线 Boot Loader 的全过程。
20. 为什么在第 6.4.3 节中的程序没有加入中断向量表？
21. 程序设计练习：自己建立一个 Boot 表，并将这个 Boot 表写入到 FLASH 中。
22. 如何查看写入 FLASH 中的值?
23. SY-5402EVM 板采用并口 Boot 时数据区和 I/O 空间的 0xFFFF 地址是否重合？

第七章

一个完整实例

7.1 本章内容简介

本章是在程序设计实例这个层次上对前面六章内容的一个总结。这个总结体现在以下三个方面：其一，在设计思路上，本章实例是按照 DSP 应用系统软件设计的流程来操作的，首先是考虑 DSP 应用系统要完成的功能，然后按系统的初始化、算法设计、Boot Loader 等一步一步将本项目完成；其二，在开发过程中，本实例给出了分析 DSP 应用系统硬件的方法，强调了如何使用 C/C++语言开发一个 DSP 应用系统软件；其三，在具体实现上，借助了 SY-5402EVM 板来实现本实例的设计，并给出了设计的全部工程文件源代码，充分验证了设计的可行性。

本章在介绍程序设计实例时注重了其完整性和全面性，以便读者对 DSP 应用系统程序设计过程有一个完整的全面的了解。

思考题

(1) 开发一个 DSP 应用系统软件的总体过程是什么样的？

(2) 本章实例中哪些语句是时序编程？哪些是数据编程？

7.2 DSP/BIOS 编程实例

1) 准备工作

使用一条音频线将计算机的音频输出端与 SY-5402EVM 的音频输入端相连通，SY-5402EVM 板的 CLKMD1、CLKMD2、CLKMD3 三个跳线设置为 110，将 JP12、JP13、JP14、JP15 四个跳线均跳低，MP/MC 跳线跳低，JP1 跳线跳高，用一个耳机连接到 SY-5402EVM 板的音频输出端。将 SY-5402EVM 板与仿真器连接好，仿真器与计算机连接好。打开计算机、仿真器和 SY-5402EVM 板的电源，在 CCStudio-Setup 中配置仿真目标板并进入程序开发主窗口。

2) 开发过程

本实例开发经过两个过程：其一是开发 DSP 应用程序，这个应用程序是被下载到 DSP 功能板上的 FLASH 中的；其二是开发 DSP 的在线 Boot Loader 程序，这个程序是在 CCStudio 中运行的，将 DSP 应用程序下载到 DSP 功能板上的 FLASH 中，这个程序本身保留在计算机硬盘上。DSP 应用程序的开发又分为系统初始化和算法设计两部分，这里的算法是在 VC5402 内部对输入的语音信号做了自适应增益控制之后，再送出到耳机。这里给出的是一个通用数字信号处理器中实现自适应增益控制的算法实例，是典型的数据编程。

3) 实现功能

本实例要实现的功能是音频扩放系统的自动增益控制（AGC）。计算机通过媒体播放器向 SY-5402EVM 板送出音频信号，通过连接到 SY-5402EVM 板音频输出端的耳机听到清晰的音乐。调整计算机输出音量的大小，在耳机中听到的声音音量的大小几乎保持不变。

7.3 实例设计过程

7.3.1 系统初始化

本实例运行在 SY-5402EVM 板上，本节的系统初始化包括两部分：其一是 VC5402 的初始化，其二是 TLC320AD50C 的初始化。

对于 VC5402 的初始化，又分为三步：第一步是将 VC5402 的工作时钟调整到 100 MHz。VC5402 上电硬复位后，按照 CLKMD1、CLKMD2 和 CLKMD3 的设置（110）而工作在 PLL×1 模式下，即工作在与外部晶振频率相同的频率上。这时，VC5402 的工作时钟为 10 MHz。所以 VC5402 的初始化第一步是将时钟从 10 MHz 倍频到 100 MHz。第二步是 VC5402 的 CPU 控制寄存器的初始化。第三步是 VC5402 的串行口 1 的初始化。

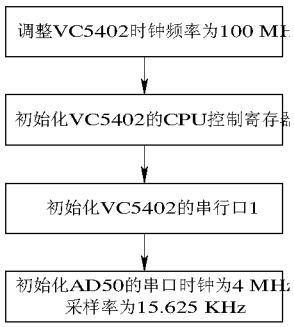


图 7-1 系统初始化流程图

系统初始化的流程图如图 7-1 所示。具体写入的控制字含义请参考本章程序注解和在线帮助。

7.3.2　读写串口

在 SY-5402EVM 上，从 AD50 读入数据和向 AD50 发送数据都是通过读写 VC5402 的串口 1 来实现的。关于这个读写方法和过程在前面的章节中多次讲过，这里不再冗述。

7.3.3　自动增益控制（AGC）

本节将详细介绍程序中的自动增益控制算法。自动增益控制有很多种方法，本章实例中采用了峰值自动增益控制算法，它的算法流程框图如图 7-2 所示。

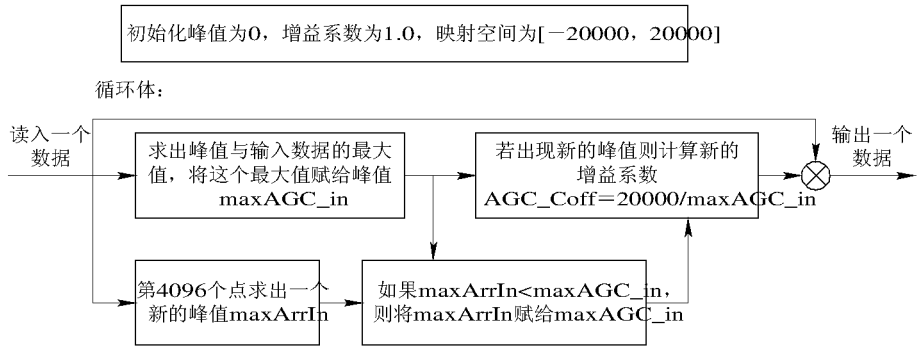


图 7-2　自动增益控制的算法流程图

从串口 1 读入的音频数据是 16 位的整型数，一般来说，这些数据是很小的，通过上面的 AGC 将输入的音频数据投影在[-20000，20000]区间内，从而使得不论输入的数据点数值的大小，都会等比例地向这个空间投射。从图 7-2 可以看出，增益系数是相对稳定的，当向上调节音量时，峰值会自动增加，而使得增益系数下降；当向下调节音量时，新的峰值会明显小于原来的峰值而取代原来的峰值，从而使得峰值下降，而增益系数上升。最后输出的数据为输入的数据乘以这个增益系数而投影到[-20000，20000]区间上的数据。实际程序设计时，还要考虑到输入数据的正负号，为了节约指令，这里没有用到取绝对值符号。

7.3.4　在线 Boot Loader

DSP 应用程序设计完成后，下一步工作就是将这个应用程序写入到 SY-5402EVM 板上的 FLASH 中，这个过程称为在线写 FLASH 或称在线 Boot Loader。在第六章中我们已经详细地介绍了这种编程方法，本章中将不再重述。

7.4　程序源代码及注解

7.4.1　DSP 应用程序源代码

图 7-3 为 DSP 应用程序工程文件 agc_audio.pjt 的管理窗口，图 7-4 为本程序的“裁缝师”测试窗口。

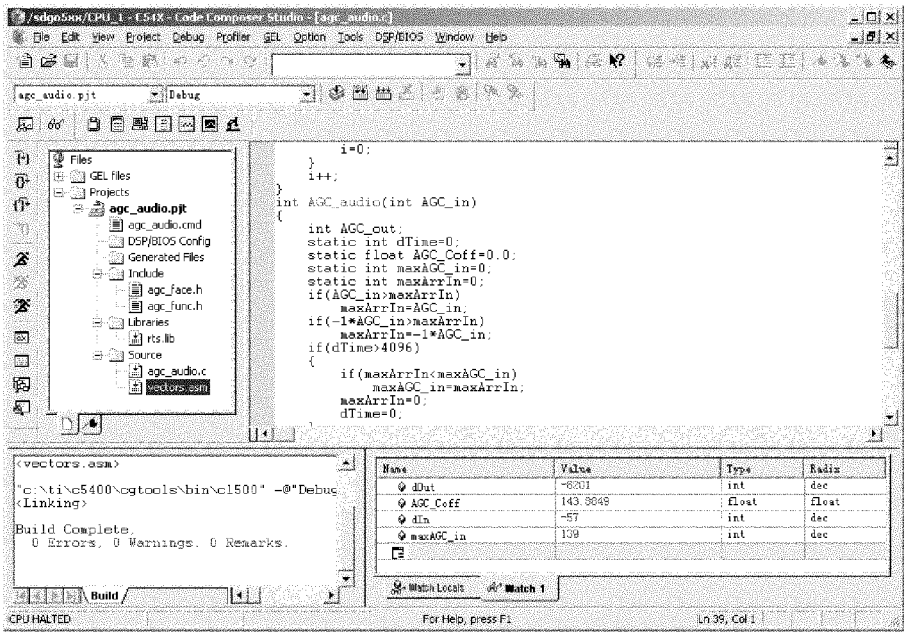


图 7-3　工程文件管理器主窗口

从图 7-4 中可以看出，工程文件中包括了 agc_audio.c、vectors.asm、agc_face.h、agc_func.h、agc_audio.cmd 和 rts.lib 文件。从图 7-4 中可以推算出对于单一数据点，程序的运行时间小于 6 ms，程序适合于强实时性的场合。

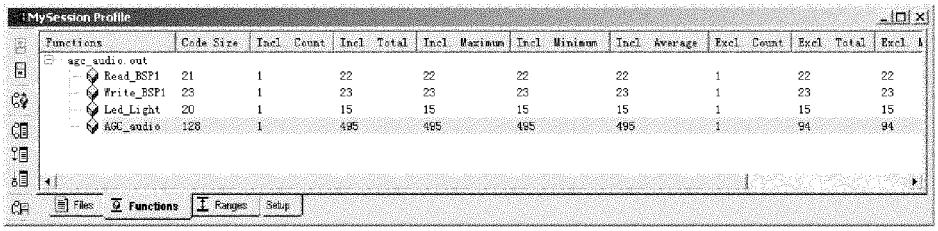


图 7-4 “裁缝师”窗口

下面为主程序文件 agc_audio.c 的源程序代码：

```
/*****************************************************************************/
/* Filename: agc_aduio.c                                                     */
/* Author:ZhangYong                                                          */
/* 2002-12                                                                   */
/*****************************************************************************/

#include "agc_face.h"
#include "agc_func.h"
```

```
int   dIn,dOut;
int   Read_BSP1(void);                    //读串口 1
void Write_BSP1(int);                     //写串口 1
void Led_Light(void);                     //D3-LED 灯闪烁
int   AGC_audio(int);                     //自动增益控制函数
// Main program
void main()
{
     init_board();                        //VC5402 和 AD50 的初始化
     while (1)
     {
          dIn=Read_BSP1();                //从串行口 1 读入一个数据
          dOut=AGC_audio(dIn);            //做自动增益控制
          Write_BSP1(dOut);               //将数据送出到 AD50
          Led_Light();                    //点亮或关闭 LED 灯
      }
}

int Read_BSP1(void)                       //从串行口 1 读入一个数据子函数
{
     int dDataIn;
     *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR;   //Receive Data from McBSP1
     while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
     dDataIn=*(volatile u16*)DRR1_ADDR(1);

     return dDataIn;
}
void Write_BSP1(int dDataOut)             //写一个数据到串口 1 的子函数中
{
     *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;   //Transmit Data To McBSP1
     while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
     *(volatile u16 *)DXR1_ADDR(1)=dDataOut;
}
void Led_Light(void)        //LED 灯的闪烁子函数，注意这里面定义的静态变量的用法
{
     static int i=0;
     if(i>1024)
     {
     //      *(int *)(0x0007)^=0x2000;
```

```
            *(volatile u16 *)reg_ST1 ^= 0x2000;     //Every 1024 times XF changed
            i=0;
        }
        i++;
    }
    int AGC_audio(int AGC_in)        //自动增益控制子函数，注意定义的静态变量
    {
        int AGC_out;
        static int dTime=0;
        static float AGC_Coff=0.0;
        static int maxAGC_in=0;
        static int maxArrIn=0;
        if(AGC_in>maxArrIn)
            maxArrIn=AGC_in;
        if(-1*AGC_in>maxArrIn)
            maxArrIn=-1*AGC_in;
        if(dTime>=4096)
        {
            if(maxArrIn<maxAGC_in)
                maxAGC_in=maxArrIn;
            maxArrIn=0;
            dTime=0;
        }
        dTime++;                          //上面语句为每 4096 个点计算一个新的峰值

        if(AGC_in > maxAGC_in)
        {
            maxAGC_in=AGC_in;
            if(maxAGC_in>1)
                AGC_Coff=20000.0/maxAGC_in;
        }
        if((-1*AGC_in)>maxAGC_in)
        {
            maxAGC_in=-1*AGC_in;
            if(maxAGC_in>1)
                AGC_Coff=20000.0/maxAGC_in;
        }    //上面两个 IF 语句是分正、负情况来判断的，计算正的峰值和正的增益系数
        AGC_out=AGC_in*AGC_Coff;          //计算输出数据
        if(AGC_out>20000)
```

```
        AGC_out=20000;
    if(AGC_out<-20000)
        AGC_out= -20000;                    //超界的判断
    return AGC_out;
}
```

//下面为头文件 agc_face.h 的内容：

```
//Filename: agc_face.h

typedef unsigned int u16;
typedef signed int s16;
//此文件包括寄存器的地址和值，前面也多次使用，具体含义请参考在线“帮助”
/* Register Definition   MCBSP   */
#define SPSA_ADDR(port)          (port ? 0x48 : 0x38)
#define SPSD_ADDR(port)          (port ? 0x49 : 0x39)
#define DRR2_ADDR(port)          (port ? 0x40 : 0x20)
#define DRR1_ADDR(port)          (port ? 0x41 : 0x21)
#define DXR2_ADDR(port)          (port ? 0x42 : 0x22)
#define DXR1_ADDR(port)          (port ? 0x43 : 0x23)

#define MCBSP_ACCSUB_ADDR(port) (port ? 0x49 : 0x39)

#define    SPCR1_SUBADDR         0x00
#define    SPCR2_SUBADDR         0x01
#define    RCR1_SUBADDR          0x02
#define    RCR2_SUBADDR          0x03
#define    XCR1_SUBADDR          0x04
#define    XCR2_SUBADDR          0x05
#define    SRGR1_SUBADDR         0x06
#define    SRGR2_SUBADDR         0x07
#define    MCR1_SUBADDR          0x08
#define    MCR2_SUBADDR          0x09
#define    RCERA_SUBADDR         0x0A
#define    RCERB_SUBADDR         0x0B
#define    XCERA_SUBADDR         0x0C
#define    XCERB_SUBADDR         0x0D
#define    PCR_SUBADDR           0x0E
```

```
#define     bsp_SPCR11      0x0021
#define     bsp_SPCR21      0x0201
#define     bsp_PCR1        0x000C
#define     bsp_RCR11       0x0040
#define     bsp_RCR21       0x0000
#define     bsp_XCR11       0x0040
#define     bsp_XCR21       0x0000
#define     bsp_SRGR11      0x0000
#define     bsp_SRGR21      0x0000
#define     bsp_MCR11       0x0000
#define     bsp_MCR21       0x0000
//---------CPU-----------
//ST1       addr:0x0007     Status register 1   ST1[13]=XF
#define     reg_ST1         0x0007
//#define   ST0             *(volatile unsigned int*)0x06
//#define   ST0_ADDR        0x06

#define     PMST        0x001D
#define     SWWSR       0x0028
#define     SWCR        0x002B
#define     BSCR        0x0029
#define     CLKMD       0x0058

#define     PMST_VAL        0x00A0  //interupt vectors from ox80
#define     SWWSR_VAL       0x7fff  //这个设置不是最优的
#define     SWCR_VAL        0x0001
#define     BSCR_VAL        0x8802
#define     CLKMD_VAL       0x9807
```

//下面为头文件 agc_func.h 的内容：

```
/*Filename: agc_func.h*/
void init_ad50(void)
{
    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;   //Transmit Data 0x0001 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0001;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;   //Transmit Data 0x0450 To AD50
```

```
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0450;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;    //Transmit Data 0x0001 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0001;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;    //Transmit Data 0x0210 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0210;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;    //Transmit Data 0x0001 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0001;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;    //Transmit Data 0x0320 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0320;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;    //Transmit Data 0x0001 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0001;

    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;    //Transmit Data 0x0101 To AD50
    while(!((*(volatile u16 *)SPSD_ADDR(1)) & 0x0002)){};
    *(volatile u16 *)DXR1_ADDR(1)=0x0101;
}

void init_board(void)          //下面的初始化已多次提到，这里不再注解
{
     *(volatile u16 *)CLKMD = 0x0000;
     while(*(volatile u16 *)CLKMD & 0x0001){};
     *(volatile u16 *)CLKMD = CLKMD_VAL;

     *(volatile u16 *)PMST = PMST_VAL;
     *(volatile u16 *)SWWSR =SWWSR_VAL;
     *(volatile u16 *)SWCR =SWCR_VAL;
     *(volatile u16 *)BSCR =BSCR_VAL;
```

```
    *(volatile u16 *)SPSA_ADDR(1)=SPCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SPCR11;
    *(volatile u16 *)SPSA_ADDR(1)=SPCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SPCR21;
    *(volatile u16 *)SPSA_ADDR(1)=RCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_RCR11;
    *(volatile u16 *)SPSA_ADDR(1)=RCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_RCR21;
    *(volatile u16 *)SPSA_ADDR(1)=XCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_XCR11;
    *(volatile u16 *)SPSA_ADDR(1)=XCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_XCR21;
    *(volatile u16 *)SPSA_ADDR(1)=SRGR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SRGR11;
    *(volatile u16 *)SPSA_ADDR(1)=SRGR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_SRGR21;
    *(volatile u16 *)SPSA_ADDR(1)=MCR1_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_MCR11;
    *(volatile u16 *)SPSA_ADDR(1)=MCR2_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_MCR21;
    *(volatile u16 *)SPSA_ADDR(1)=PCR_SUBADDR;
    *(volatile u16 *)SPSD_ADDR(1)=bsp_PCR1;

    init_ad50();            //AD50 的初始化
}
```

//下面为配置文件 agc_audio.cmd 的内容：

```
-m   agc_audio.map        /*通过这个 MAP 表查看存储空间分配情况*/

MEMORY
{
    PAGE 0:   PROG(RWXI):   origin=0x100,   length=0x3000
              VECS(RXI):    origin=0x80,    length=0x80
    PAGE 1:   DATA(RWI):    origin=0x3100,  length=0xF00
}

SECTIONS
{
```

```
        .text       {} >        PROG      PAGE 0
        .cinit      {} >        PROG      PAGE 0
        .pinit      {} >        PROG      PAGE 0
        .vectors    {} >        VECS      PAGE 0
        .stack      {} >        DATA      PAGE 1
        .bss        {} >        DATA      PAGE 1
        .const      {} >        DATA      PAGE 1
        .switch     {} >        DATA      PAGE 1
        .sysmem     {} >        DATA      PAGE 1
        .cio        {} >        DATA      PAGE 1
        .far        {} >        DATA      PAGE 1
}
```

//下面为中断向量文件 vectors.asm 的内容：

```
        .sect ".vectors"
        .ref _c_int00                ; 定义在 rts.lib 库中
        .align   0x80                ; 中断向量表必须占有 128 字的整倍数区块

RESET:
        BD       _c_int00
        STM      #128, SP            ; 按实际情况分堆栈的大小，一般 0x200 就足够了
nmi:    RETE                         ; 必须占有 4 个字的空间
        NOP
        NOP
        NOP
        .end
```

7.4.2　在线 Boot Loader 程序源代码

图 7-5 展示了 Boot Loader 程序工程文件的组成。图 7-6 为运行完毕 Boot Loader 程序后，观察 DATA 空间或是 I/O 空间的地址 0x8000 开始处的内容，即写入到 FLASH 中的内容。因为对于 SY-5402EVM 板，FLASH 的 0x8000 至 0xFFFF 区段同时映射到了 DATA 空间和 I/O 空间，所以可以通过查看这两个空间的内容的方法来验证 FLASH 编程的正确性。

从图 7-5 中可以看出，Boot Loader 工程文件包括 boot_5402.c、boot_5402.cmd、boot_5402.h、boot_data.h、port_face.h、port_func.h 和 rts.lib 库文件。

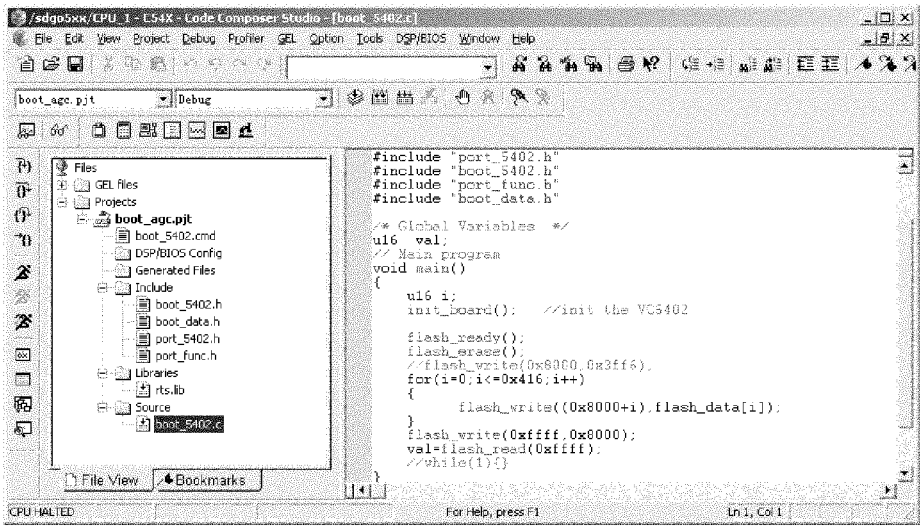


图 7-5　Boot Loader 程序的 CCStudio 主窗口

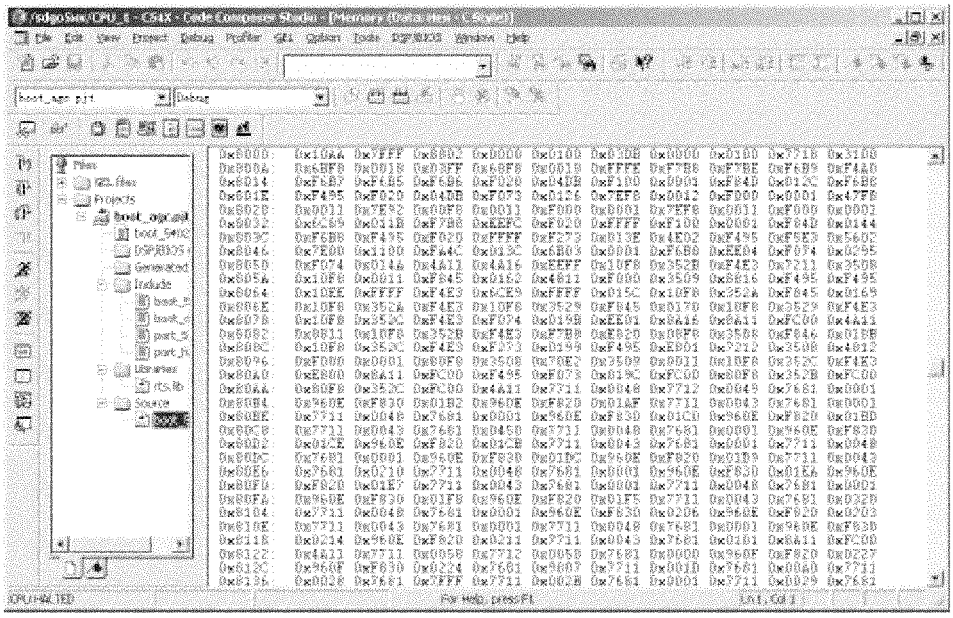


图 7-6　FLASH 中从 0x8000 地址处开始的内容

下面首先介绍 boot_data.h 文件生成的方法：

在 DSP 应用程序设计时，使用-v548 编译选项，生成一个 Release 版的可执行 agc_audio.out 文件；编写一个 Coff2Hex.cmd 文件用 hex500 可执行文件生成 boot 表，下面列出了 Coff2Hex.cmd 文件的源代码；编写一个 C/C++通用程序将这个 Boot 表转换为数组的形式，存放在 boot_data.h 中。

/*下面为 Coff2Hex.cmd 文件的内容*/

```
/* Coff2Hex.cmd*/
agc_audio.out    /* hex500 user_audio.cmd*/
-a
-map    agc_audio.map
-o agc_audio.hex
-bootorg PARALLEL
-e 0x100    /* -e    _c_int00*/
-boot
-swwsr 0x7fff                    /*SWWSR 寄存器的值不是最优的*/
-bscr 0x8802
-memwidth 16/* default 16*/
-romwidth 16
```

在命令方式下，执行下面的命令可以生成 Boot 表文件：

```
hex500    Coff2Hex.cmd
```

注意：必须加上扩展名.cmd。

编写一个通用的 C/C++程序将这个 Boot 表转换为下面的数据头文件 boot_data.h，并包括在工程文件中。

写入到 FLASH 中的数据头文件 boot_data.h 的内容如下：

```
/* The data to be written to FLASH */
u16 flash_data[1047]={
0x10AA,0x7FFF,0x8802,0x0000,0x0100,0x03DB,0x0000,0x0100,0x7718,0x3100,0x6BF8,
0x0018,0x03FF,0x68F8,0x0018,0xFFFE,0xF7B8,0xF7BE,0xF6B9,0xF4A0,0xF6B7,0xF6B5,
0xF6B6,0xF020,0x04DB,0xF100,0x0001,0xF84D,0x012C,0xF6B8,0xF495,0xF020,0x04DB,
0xF073,0x0126,0x7EF8,0x0012,0xF000,0x0001,0x47F8,0x0011,0x7E92,0x00F8,0x0011,
0xF000,0x0001,0x7EF8,0x0011,0xF000,0x0001,0x6C89,0x011B,0xF7B8,0xEEFC,0xF020,
0xFFFF,0xF100,0x0001,0xF84D,0x0144,0xF6B8,0xF495,0xF020,0xFFFF,0xF273,0x013E,
0x4E02,0xF495,0xF5E3,0x5602,0x7E00,0x1100,0xFA4C,0x013C,0x6B03,0x0001,0xF6B8,
0xEE04,0xF074,0x0295,0xF074,0x014A,0x4A11,0x4A16,0xEEFF,0x10F8,0x352B,0xF4E3,
0x7211,0x3508,0x10F8,0x0011,0xF845,0x0162,0x4811,0xF000,0x3509,0x8816,0xF495,
0xF495,0x10EE,0xFFFF,0xF4E3,0x6CE9,0xFFFF,0x015C,0x10F8,0x352A,0xF845,0x0169,
0x10F8,0x352A,0xF4E3,0x10F8,0x3529,0xF845,0x0170,0x10F8,0x3529,0xF4E3,0x10F8,
0x352C,0xF4E3,0xF074,0x019B,0xEE01,0x8A16,0x8A11,0xFC00,0x4A11,0x8811,0x10F8,
0x352B,0xF4E3,0xF7B8,0xE820,0x08F8,0x3508,0xF846,0x018B,0x10F8,0x352C,0xF4E3,
0xF273,0x0199,0xF495,0xE801,0x7212,0x3508,0x4812,0xF000,0x0001,0x80F8,0x3508,
```

0x70E2,0x3509,0x0011,0x10F8,0x352C,0xF4E3,0xE800,0x8A11,0xFC00,0xF495,0xF073,
0x019C,0xFC00,0x80F8,0x352B,0xFC00,0x80F8,0x352C,0xFC00,0x4A11,0x7711,0x0048,
0x7712,0x0049,0x7681,0x0001,0x960E,0xF830,0x01B2,0x960E,0xF820,0x01AF,0x7711,
0x0043,0x7681,0x0001,0x7711,0x0048,0x7681,0x0001,0x960E,0xF830,0x01C0,0x960E,
0xF820,0x01BD,0x7711,0x0043,0x7681,0x0450,0x7711,0x0048,0x7681,0x0001,0x960E,
0xF830,0x01CE,0x960E,0xF820,0x01CB,0x7711,0x0043,0x7681,0x0001,0x7711,0x0048,
0x7681,0x0001,0x960E,0xF830,0x01DC,0x960E,0xF820,0x01D9,0x7711,0x0043,0x7681,
0x0210,0x7711,0x0048,0x7681,0x0001,0x960E,0xF830,0x01EA,0x960E,0xF820,0x01E7,
0x7711,0x0043,0x7681,0x0001,0x7711,0x0048,0x7681,0x0001,0x960E,0xF830,0x01F8,
0x960E,0xF820,0x01F5,0x7711,0x0043,0x7681,0x0320,0x7711,0x0048,0x7681,0x0001,
0x960E,0xF830,0x0206,0x960E,0xF820,0x0203,0x7711,0x0043,0x7681,0x0001,0x7711,
0x0048,0x7681,0x0001,0x960E,0xF830,0x0214,0x960E,0xF820,0x0211,0x7711,0x0043,
0x7681,0x0101,0x8A11,0xFC00,0x4A11,0x7711,0x0058,0x7712,0x0058,0x7681,0x0000,
0x960F,0xF820,0x0227,0x960F,0xF830,0x0224,0x7681,0x9807,0x7711,0x001D,0x7681,
0x00A0,0x7711,0x0028,0x7681,0x7FFF,0x7711,0x002B,0x7681,0x0001,0x7711,0x0029,
0x7681,0x8802,0x7711,0x0048,0x7681,0x0000,0x7711,0x0049,0x7681,0x0021,0x7711,
0x0048,0x7681,0x0001,0x7711,0x0049,0x7681,0x0201,0x7711,0x0048,0x7681,0x0002,
0x7711,0x0049,0x7681,0x0040,0x7711,0x0048,0x7681,0x0003,0x7711,0x0049,0x7681,
0x0000,0x7711,0x0048,0x7681,0x0004,0x7711,0x0049,0x7681,0x0040,0x7711,0x0048,
0x7681,0x0005,0x7711,0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,0x0006,0x7711,
0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,0x0007,0x7711,0x0049,0x7681,0x0000,
0x7711,0x0048,0x7681,0x0008,0x7711,0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,
0x0009,0x7711,0x0049,0x7681,0x0000,0x7711,0x0048,0x7681,0x000E,0x7711,0x0049,
0x7681,0x000C,0xF074,0x01A5,0x8A11,0xFC00,0xEEFF,0xF074,0x021A,0xF074,0x02A6,
0x80F8,0x3500,0xF074,0x02DE,0x80F8,0x3501,0xF074,0x02B8,0xF074,0x02CA,0xF073,
0x0298,0x4A11,0x7711,0x0048,0x7712,0x0049,0x7681,0x0000,0x960E,0xF830,0x02B3,
0x960E,0xF820,0x02B0,0x7711,0x0041,0x1081,0x8A11,0xFC00,0x4A11,0x7711,0x0048,
0x7712,0x0049,0x7681,0x0001,0x960E,0xF830,0x02C5,0x960E,0xF820,0x02C2,0x7711,
0x0043,0x8081,0x8A11,0xFC00,0xF7B8,0xF020,0x0400,0x4A11,0x08F8,0x3502,0xF842,
0x02D9,0x7711,0x0007,0x6A81,0x2000,0x76F8,0x3502,0x0000,0x6BF8,0x3502,0x0001,
0x8A11,0xFC00,0x4A11,0xF7B8,0x8811,0x10F8,0x0008,0x08F8,0x3507,0xFA47,0x02EB,
0xF495,0xEEFC,0x7311,0x3507,0x4811,0xF584,0x10F8,0x000B,0x08F8,0x3507,0xF847,
0x02F5,0x81F8,0x3507,0xF020,0x1000,0x08F8,0x3503,0xF842,0x030A,0x10F8,0x3506,
0x08F8,0x3507,0xF847,0x0304,0x71F8,0x3507,0x3506,0x76F8,0x3507,0x0000,0x76F8,
0x3503,0x0000,0x4811,0x10F8,0x0008,0x6BF8,0x3503,0x0001,0x08F8,0x3506,0xF847,
0x0328,0x4811,0x10F8,0x0008,0xF010,0x0002,0xFA43,0x0328,0x7311,0x3506,0xF274,
0x043D,0xF495,0x4811,0x4E00,0xF274,0x0366,0x56F8,0x352E,0x4EF8,0x3504,0xF274,
0x043D,0xF495,0x4811,0xF7B8,0x4E02,0x10F8,0x000B,0x08F8,0x3506,0xF847,0x0348,
0x4811,0x10F8,0x0008,0xF010,0xFFFE,0xFA46,0x0348,0x81F8,0x3506,0xF274,0x043D,

0xF495,0xF640,0x4E00,0xF274,0x0366,0x56F8,0x352E,0x4EF8,0x3504,0x56F8,0x3504,
0x4E00,0x5602,0xF074,0x0468,0xF074,0x0401,0xF7B8,0x11F8,0x0008,0xF310,0x4E20,
0xF84F,0x035B,0xF273,0x0363,0xF020,0x4E20,0x11F8,0x0008,0xF310,0xB1E0,0xF84A,
0x0363,0xF020,0xB1E0,0xEE04,0x8A11,0xFC00,0xF495,0x4A0B,0x4A0C,0x4A0D,0x4A0E,
0xF6B7,0xEEF3,0xF7B8,0xF468,0xF478,0xF845,0x03D3,0x6F05,0x0C79,0x800B,0xF063,
0x007F,0xF060,0x0080,0x820A,0x5612,0xF845,0x03F1,0x6F06,0x0C79,0x8009,0xF063,
0x007F,0xF060,0x0080,0x8208,0x1005,0x1C06,0xF030,0x0100,0x8004,0x1005,0xF030,
0x00FF,0x1106,0xF330,0x00FF,0xF000,0x007F,0xF620,0x8001,0xF843,0x03E9,0xF010,
0x00FF,0xF846,0x03D9,0x560A,0xF0E6,0x5708,0xF3E7,0x4F08,0xEC0E,0x1E08,0x8002,
0x0A02,0xEC0A,0x1E08,0x6F03,0x0C85,0x3002,0x2409,0xF0FF,0xEC0B,0x1E08,0xF0E4,
0xF030,0xFFFF,0xF0E1,0xF484,0x0203,0x3C02,0x1101,0xF48E,0xF495,0xF48F,0x8C01,
0x0901,0xF0FF,0xF300,0x0001,0xF000,0x0020,0xF48E,0xF495,0xF48F,0x8C01,0x0901,
0x8101,0xF84F,0x03E9,0xF310,0x00FF,0xF84A,0x03D9,0xF0F9,0x8003,0xF038,0x7F00,
0x8202,0x4404,0x3C01,0xF0E7,0x5002,0xEE0D,0x8A0E,0x8A0D,0x8A0C,0x8A0B,0xFC00,
0x7600,0x0002,0xF074,0x0400,0xF483,0xF061,0x0081,0x4504,0xF84C,0x03D3,0xF162,
0x8000,0xF273,0x03D3,0xF2C0,0xF495,0x7600,0x0001,0xF074,0x0400,0xF273,0x03D3,
0xF420,0xF495,0x7600,0x0003,0xF074,0x0400,0x4505,0xF363,0x0100,0xF061,0x0081,
0xF065,0x8000,0xF273,0x03D3,0xF2A7,0xF495,0xFC00,0xF495,0x4A0B,0x4A0C,0x4A0D,
0x4A0E,0xF6B8,0xEEFD,0xF845,0x0423,0x6F01,0x0C79,0xF0E9,0xF0F7,0xF060,0x0080,
0x6F02,0x0C67,0x1101,0xF330,0x00FF,0xF310,0x007F,0xF84B,0x0429,0xF310,0x000E,
0xF84E,0x042D,0x890E,0x6101,0x0100,0x1402,0xFD30,0xF484,0xEE03,0x8A0E,0x8A0D,
0x8A0C,0x8A0B,0xFC00,0xF273,0x0423,0xF420,0xF495,0x7600,0x0002,0xF074,0x0400,
0x6101,0x0100,0xF020,0x7FFF,0xF820,0x0423,0xF000,0x0001,0xF273,0x0423,0xF484,
0xF495,0xF495,0x4A0E,0x4A0B,0x4A0C,0x4A0D,0xF7B8,0xEEFD,0xF845,0x045E,0xF46F,
0xF469,0xF478,0x8201,0xF6B9,0xF4BA,0xF485,0xF870,0x0464,0xF48E,0xE98D,0x8C00,
0x0900,0xF48F,0xF0E2,0xF0FE,0xF0F0,0xF2AE,0x6101,0x8000,0xF0E9,0xFF30,0xF064,
0x8000,0xEE03,0x8A0D,0x8A0C,0x8A0B,0x8A0E,0xFC00,0xF273,0x045E,0xF062,0xC700,
0xF495,0x4A0B,0x4A0C,0x4A0D,0x4A0E,0xF6B7,0xEEF3,0xF7B8,0xF468,0xF478,0xF845,
0x04C1,0x6F05,0x0C79,0x800B,0xF063,0x007F,0xF060,0x0080,0x820A,0x5612,0xF845,
0x04C1,0x6F06,0x0C79,0x8009,0xF063,0x007F,0xF060,0x0080,0x8208,0x1005,0x1C06,
0xF030,0x0100,0x8004,0x1005,0xF030,0x00FF,0x1106,0xF330,0x00FF,0xF310,0x007F,
0xF600,0x8001,0xF843,0x04D3,0xF010,0x00FF,0xF846,0x04C7,0x300B,0x2409,0x2508,
0xF510,0x300A,0x2409,0xF600,0x2508,0x8102,0x3C02,0xF000,0x0040,0xF468,0xF478,
0xF6B8,0x1101,0xF842,0x04B2,0xF0FF,0xF300,0x0001,0x8101,0xF84F,0x04D3,0xF310,
0x00FF,0xF84A,0x04C7,0xF0F9,0x8003,0xF038,0x7F00,0x8202,0x4404,0x3C01,0xF0E7,
0x5002,0xEE0D,0x8A0E,0x8A0D,0x8A0C,0x8A0B,0xFC00,0x7600,0x0002,0xF074,0x0400,
0x4504,0xF020,0xFFFF,0xF2A7,0xF273,0x04C1,0xF060,0x7F7F,0x7600,0x0001,0xF074,
0x0400,0xF273,0x04C1,0xF420,0xF495,0x0020,0x0000,0x04DB,0x0001,0x3508,0x0000,
0x0001,0x3529,0x0000,0x0001,0x352A,0x0000,0x0001,0x352B,0x019E,0x0001,0x352C,

```
0x019E,0x0001,0x3502,0x0000,0x0001,0x3503,0x0000,0x0002,0x3504,0x0000,0x0000,
0x0001,0x3506,0x0000,0x0001,0x3507,0x0000,0x0000,0x0008,0x0000,0x0080,0xF273,
0x0100,0x7718,0x0080,0xF4EB,0xF495,0xF495,0xF495,0x0002,0x0000,0x352E,0x469C,
0x4000,0x0000};
//0x0000-0x0416 To FLASH 0x8000--0x8416
```

下面列出了其他源文件的内容：

主程序 boot_agc.c 的内容：

```
/*****************************************************************************/
/* filename: boot_agc.c                                                      */
/* Author:ZhangYong                                                          */
/* 2002-12                                                                   */
/*****************************************************************************/
#include "port_5402.h"
#include "boot_5402.h"
#include "port_func.h"
#include "boot_data.h"

/* Global Variables   */
u16   val;
// Main program
void main()
{
      u16 i;
      init_board();       //init the VC5402

      flash_ready();                  //将 FLASH 设置为正常读写状态
      flash_erase();                  //将 FLASH 整片擦除
      //flash_write(0x8000,0x3ff6);
       for(i=0;i<=0x416;i++)          //将文件 boot_data.h 中的数据写入到 FLASH 中
       {
              flash_write((0x8000+i),flash_data[i]);
       }
       flash_write(0xffff,0x8000);  //写 0x8000 到 FLASH 的 0xFFFF 地址处
       val=flash_read(0xffff);       //读出 FLASH 的 0xFFFF 地址处的值并验证一下
       //while(1){}
}
```

配置文件 boot_5402.cmd 的内容：

```
/*-m    boot_5402.map*/

MEMORY
{
      PAGE 0:   PROG(RWXI):   origin=0x100,    length=0x3000
                VECS(RXI):    origin=0x80,     length=0x80
      PAGE 1:   DATA(RWI):    origin=0x3100,   length=0xF00
}

SECTIONS
{
      .text       {} >        PROG      PAGE 0
      .cinit      {} >        PROG      PAGE 0
      .pinit      {} >        PROG      PAGE 0
      .vectors    {} >        VECS      PAGE 0
      .stack      {} >        DATA      PAGE 1
      .bss        {} >        DATA      PAGE 1
      .const      {} >        DATA      PAGE 1
      .switch     {} >        DATA      PAGE 1
      .sysmem     {} >        DATA      PAGE 1
      .cio        {} >        DATA      PAGE 1
      .far        {} >        DATA      PAGE 1
      .data       {} >        DATA      PAGE 1
}
```

其他的三个头文件 boot_5402.h、port_5402.h、port_func.h 均在第六章中列出了，这里不再重复。

这个程序中没有加入中断向量文件，这是因为这个文件是借助 CCStudio 集成环境在线写 FLASH 用的，它没有必要下载到 DSP 功能板上，所以我们省略了中断向量文件。

7.5　本 章 小 结

本章总结了进行 DSP 应用系统开发的软件设计的全过程。概括地讲，其过程是：首先，编写 DSP 应用程序，并生成 Release 版的 agc_audio.out；然后，生成 Boot 表文件；最后，编写在线 Boot Loader 程序，将 Boot 表在线写入 FLASH 中。完成以上过程后，可以将 SY-5402EVM 板与仿真器断开连接，单独给 SY-5402EVM 板上电，可以得到预期的功能。此外，本章也给出了“裁缝师”测试的结果，以资证明本 C 程序的可行性。

习 题 七

1. 开发 DSP 应用系统软件的总体过程是怎样的？
2. 结合本章的初始化程序说明如何将 VC5402 的 CPU 时钟调整到 100 MHz。
3. 结合本章的初始化程序说明各寄存器的值的含义。
4. 简述如何对 TLC320AD50 进行初始化。
5. 简述本章中使用的 AGC 的原理。
6. 介绍“裁缝师”窗口中各栏目的含义（参考图 7-4）。
7. 编写一个通用的 C/C++程序，将 Boot 表转换为 boot_data.h 文件中的内容。
8. 找出本章程序中所有的时序编程语句。
9. 编写一种新的在线 Boot Loader 程序,直接从硬盘上读取 Boot,表并写入到 FLASH 中。
10. 编写另一种 AGC 算法实现本章程序中的 AGC 功能。

附录　相关术语表

A

Absolute Addressing　绝对寻址
Accumulator　累加器
Accumulator Addressing　累加器寻址
Address Bus　地址总线
ALU(Arithmetic Logic Unit)
　　算术逻辑单元
API(Application Programming Interface)
　　应用程序接口
Assembly Language　汇编语言
Auxiliary　辅助的

B

Barrel shifter　筒形移位器
Breakpoint　断点
BSCR(Bank-Switching Control Register)
　　块开关控制寄存器
BSP(Buffered Serial Port)串缓串行口
Bus Structure　总线结构

C

Call　调用
CCS(Code Composer Studio IDE)
　　代码组织平台集成环境
CCStudio(Code Composer Studio IDE)
　　代码组织平台集成环境
CodeMaestro　代码魔术师
COFF(Common Object File Format)
　　DSP 的目标文件结构
Command Window　命令窗口
Component　组件
Conditional Call　条件调用
Configuration　配置
CPU(Central Processing Unit)
　　中央处理单元
CSL(Chip Support Library)
　　片上支持库
CSSU(Compare,Select,and Store Unit)
　　比较、选择、存储单元

D

Data Converter　数据转换
DARAM(dual-access RAM)
　　双访问 RAM
Data Bus　数据总线
Debug　调试
DP(Data page pointer)　数据页指针
DMA(direct memory access)
　　直接内存访问
Direct Addressing　直接寻址
Data Type　数据类型
DSP(Digital Signal Processor)
　　数字信号处理器

E

Editing Tools　编辑工具

Emulator 仿真器
Enabling Profile Configuration
启动(或使能)裁缝师配置
EVM(Evaluation module) 评估模块
External Bus 扩展总线

F

Fixed-point 定点的

G

GEL(General Extension Language)
通用扩展语言

H

HPI(Host-Port Interface) 主机接口

I

IDE 集成开发环境
Immediate Addressing 立即寻址
Indirect 间接地
Indirect Addressing 间接寻址
Interrupt 中断

L

Load Program 装入程序

M

McBSP(Multichannel BSP)
多通道缓冲串行口
Memory 存储器
Memory-mapped Register
存储器映射寄存器
Memory-Mapped Register Addressing
存储器映射寄存器寻址
Module 模块
Multiplier/Adder Unit 乘加器

O

On-chip peripherals 片上外设
Optimization tools 优化工具
Overflow Handling 溢出控制

P

Parallel 并行
PBC(Profile Based Compiler)
编译优化裁缝师
PC(Program Counter) 程序计数器
Peripheral 外设
Pipeline 流水线
PLL(phase-locked loop) 锁相环
Plug-in 插件
PMST(Processor Mode Status Register)
处理器模式状态寄存器
Poject Management 工程(或项目)管理器
Power-Down Mode 节能模式
Probe Point 探针
Profiler 裁缝师(意译)
Project 工程(或项目)

R

RAM(Random-Access Memory)
随机存储器
Release 发行(版)
Reset 复位
ROM(Read-Only Memory)只读存储器
Run 运行
RTA(Real-Time Analysis) 实时分析器

RTDX(Real Time Data eXchange) 实时数据交换

S

SARAM(Single-Access RAM)单访问 RAM
Section 段
Serial Port 串行口
Simulator 模拟器
SP(Stack Pointer) 堆栈指针
Stack Addressing 堆栈地址
Status register 状态寄存器
SWWSR(Software Wait-State Register) 软件等待状态寄存器
Synchronous 同步
System 系统

T

TDM(Time-Division Multiplex) 时分复用
TI(Texas Instruments) 德州仪器
Trigger 触发

V

Visual Linker 可视化建立命令文件的工具

W

Watch Window 观测窗口

致　　谢

本人之所以怀着巨大的热情和强大的动力将本书完成，是因为本书的撰写得到了上海三意电子有限公司在硬件设备上的大力支持和赞助。本书所有的程序均借助了三意电子有限公司提供的 SY-5402EVM 板调试运行，使得本书成为 TMS320C5000 系列 DSP 在用 C/C++语言编程开发应用时的一本很有参考价值的书。特此向三意电子公司及彭军经理表示最诚挚的感谢。

TI 公司为本书的编写赠送了 C5000 系列光盘资料一套、C6000 资料两本、TMS320VC5410 一片；此外，TI 公司的网站是作者重要的参考资料来源。美信公司赠送了电源芯片 MAX1636EAP 两片，并曾多次及时地向作者提供各种功能的芯片。显然，只要具备了以上两个芯片，再加一个晶振及几个电阻、电容和几个接插件，就可以自行制作一个简单的仿真环境下的简易目标板。当然再有一片 FLASH 和 A/D/A 会更好一些，这个板子主要是使用片上映射存储器。在此，特向上面两家公司致以最诚挚的谢意。

广州海格通信有限公司曾为本人提供了良好的工作和实验环境，在读硕期间有幸在那里向副导师罗智勇博士学习了不少知识与经验。能在海格通信软件无线电室学习并研究 DSP 技术，为本书的问世奠定了重要的技术基础。在此，也向海格通信公司及罗博士致以最诚挚的谢意。

西安电子科技大学出版社和李惠萍编辑为本书的出版发行给予了充分的肯定和支持。特此，向西安电子科技大学出版社及李惠萍编辑表示诚挚的谢意。

这里要特别感谢我的家人，她们为我编写这本书提供了精神上的支持，她们的肯定是我长期从事本专业学习最大的动力源泉。更要感谢我的导师陈天麒教授，他为本书的出版付出了大量辛勤的工作和劳动，在百忙之中挤出时间审阅了初稿，对本书初稿提出了中肯和宝贵的意见，在此表示由衷的谢意。感谢我的几位师兄妹田达、徐催春、秦莉梅等，他们阅读了本书初稿并提出了许多宝贵意见，他们的帮助也是写成本书必不可少的一部分，在此表示最诚挚的谢意。

作者　张勇

2003 年 4 月于电子科大